Formal logic for practical use: a calculational approach

Raymond Boute

**Axioms for implication (⇒) and negation (¬)**

- **Weakening:** \( x \Rightarrow y \Rightarrow x \)
- **Distributivity:** \((x \Rightarrow y \Rightarrow z) \Rightarrow (x \Rightarrow y) \Rightarrow (x \Rightarrow z)\)
- **Contrapositive:** \((x \Rightarrow y) \Rightarrow \neg y \Rightarrow \neg x\)

**Theorems for implication**

- **Reflexivity:** \(x \Rightarrow x\)
- **Right monotonicity:** \((x \Rightarrow y) \Rightarrow (z \Rightarrow x) \Rightarrow (z \Rightarrow y)\)
- **Shunting:** \((x \Rightarrow y \Rightarrow z) \Rightarrow y \Rightarrow x \Rightarrow z\)
- **Left antimonotonicity:** \((x \Rightarrow y) \Rightarrow (y \Rightarrow z) \Rightarrow (x \Rightarrow z)\)
- **Absorption:** \((x \Rightarrow x \Rightarrow y) \Rightarrow x \Rightarrow y\)

**Theorem:** \((\neg x \Rightarrow x) \Rightarrow x\)

**Proof:**

1. \(\Rightarrow (\neg x \Rightarrow x \Rightarrow y)\)
2. \(\neg x \Rightarrow x \Rightarrow (\neg x \Rightarrow x)\)
3. \((\neg x \Rightarrow x) \Rightarrow (\neg x \Rightarrow x) \Rightarrow x\)
4. \((\neg x \Rightarrow x) \Rightarrow (\neg x \Rightarrow x) \Rightarrow x\)

**Example:**

\[\exists (x : X \land .p) \equiv \langle \text{Duality rule } \forall / \exists \rangle \neg (\forall x : X \land .r .\neg p)\]

\[\equiv \langle \text{Trading rule for } \forall \rangle \neg (\forall x : X .r \Rightarrow \neg p)\]

\[\equiv \langle a \Rightarrow b \equiv \neg a \lor b \rangle \neg (\forall x : X .\neg r \lor .\neg p)\]

\[\equiv \langle \text{De Morgan's rule} \rangle \neg (\forall x : X .\neg (r \land p))\]

\[\equiv \langle \text{Duality rule } \forall / \exists \rangle \exists (x : X .r \land .p)\]
Preface

This set of notes is a self-contained introduction to calculational logic, and is intended as background material for *Formele Semantiek* and *Formele Systeemmodellen*. It consists of selected parts from the 2001-2002 version of the course notes for *Basiswiskunde voor Computerwetenschappen*, which differs from earlier versions in the order in which axioms and theorems are introduced. Students acquainted with the earlier versions should consider this set of notes as a replacement for the corresponding chapters. This holds in particular for BC students who took *Basiswiskunde voor Computerwetenschappen* in earlier years.

For LI students, who have taken a course in formal logic in the ‘2e kandidatuur’, these notes support the transition to the *calculational* approach to logic, making it more suitable for everyday practical use in mathematical derivations.

Indeed, it has been observed (see, e.g., Gries [13]) that formal logic is often taught as just a separate branch of mathematics for its own sake, rather than as a practical tool for problem solving, theorem proving, and exposition. In non-mathematical disciplines where logic is taught as a part of the curriculum, any acquired proficiency dissipates after a short period of time, since is not maintained by other courses through application to problems of any significant logical complexity. In mathematical disciplines, logically complex problems occur regularly, but they are often handled informally. The most likely reason for this situation is that, with the traditional variants of logic, formal derivations and proofs cannot achieve the clarity and elegance associated with the algebraic style of calculation, e.g., as chained expression transformations. Perhaps this also explains why, according to a recent study, ‘novice computer science students generally have more difficulty with the concepts of logic than they have with concepts in other areas of computer science’ [1].

However, the work of Dijkstra [9] and others [12, 14] has demonstrated that the calculational approach drastically improves the convenience of formal logic, to the extent that the aforementioned impediments are removed.

The advantages of the calculational style can be further enhanced by our *functional mathematics* approach whereby various concepts that, traditionally, are not viewed or formulated as (mathematical) functions, are now captured by this single concept. The advantages become especially apparent in the predicate calculus, where predicates and quantifiers are defined as functions, and their calculation rules are derived accordingly. For instance, if $P$ is a predicate (i.e., a boolean-valued function), then $\forall P$ iff $P$ is constant with value 1.

We cover successively substitution, propositional and predicate calculus.
## Contents

**Preface** iii

0 **Expressions and Substitution** 1
   0.0 Mathematical expressions and formulas 1
   0.0.0 Conventions for syntactic definitions 1
   0.0.1 Syntax of simple mathematical expressions and formulas 2
   0.0.2 The meaning of mathematical expressions and formulas 4
   0.1 Substitution and equality 5
   0.1.0 Substitution in mathematical expressions and formulas 6
   0.1.1 Instantiation of theorems as a strict inference rule 7
   0.1.2 Equality and Leibniz’s principle 7
   0.1.3 Formal calculation with instantiation and equality 9
  0.2 Elements of the lambda calculus 11
   0.2.0 Syntax, nomenclature and conventions 11
   0.2.1 Calculation rules: substitution, equality and conversion 12
   0.2.2 Shortcuts and their application to lambda combinators 13
   0.2.3 Redexes and normal forms 14
  0.3 Variants of the lambda calculus 16
   0.3.0 Combinator calculus 16
   0.3.1 De Bruijn numbers 18
   0.3.2 Abstraction in Funmath 18

1 **Calculational Proposition Logic** 19
   1.0 Propositions, basic operators and calculation rules 19
   1.0.0 The (sub)language of propositions 19
   1.0.1 Implication and deduction rules 20
   1.0.2 Introducing the the truth constant 26
   1.0.3 Negation: axiom, calculation rules, falsehood constant 27
   1.0.4 Case analysis and a variant of Shannon expansion 28
  1.1 Introducing additional operators 29
   1.1.0 Logical equivalence as propositional equality 29
   1.1.1 Conjunction and disjunction 33
  1.2 Remarks on proofs, interpretations and equality 36
   1.2.0 Summary of proof techniques 37
   1.2.1 Nonstandard interpretations of propositional formulas 37
1.2.2 Remarks on chaining calculation steps ............... 37
1.2.3 From propositional equality to equality in general ........ 38
1.3 Binary algebra ........................................ 39
1.3.0 Binary algebra as arithmetic restricted to 0 and 1 .... 39
1.3.1 Conditionals as binary indexing ..................... 41

2 Calculating with Sets and Functions .................. 43
2.0 Calculating with sets .................................. 43
2.0.0 Set membership, Leibniz’s principle and extensionality . . . 43
2.0.1 Basic set operators and their axioms .................. 44
2.1 Calculating with functions ............................ 45
2.1.0 Function definition .................................. 45
2.1.1 Function equality .................................... 47
2.1.2 Generic functionals ................................. 48

3 Functional Predicate Calculus ......................... 51
3.0 Predicates and quantifiers: basic calculation rules ......... 51
3.0.0 Introduction: definitions and styles of expression ........ 51
3.0.1 Simple consequences of the definitions, duality .......... 52
3.0.2 Case analysis, generalized Shannon, distributivity rules . . . 53
3.1 Instantiation and generalization as metatheorems .......... 54
3.2 Completing the predicate calculus ..................... 56
3.2.0 Abstractions synthesizing common notations ............ 56
3.2.1 Wrapping up the function package .................... 58
3.2.2 Wrapping up the quantifier package ................... 59

4 Functions, Relations and Induction Principles ......... 61
4.0 Functional Mathematics ............................... 61
4.0.0 Rationale ........................................... 61
4.0.1 Using Funmath: a preview .......................... 62
4.0.2 The four syntactic constructs of Funmath ............. 63
4.0.3 Sequences and tuples as functions .................... 66
4.0.4 The generalized functional Cartesian product ........... 68
4.0.5 Elastic operators and variadic shorthands .............. 69
4.0.6 Overloading and polymorphism ....................... 75
4.1 Introduction to concrete relation calculus ............... 76
4.1.0 Relations between sets: attributes and basic operators .. 76
4.1.1 Relations on a set .................................... 77
4.1.2 Order relations ..................................... 79
4.2 Induction principles ................................. 84
4.2.0 Induction over the natural numbers ................... 84
4.2.1 Well-founded relations and induction .................. 87
Chapter 0

Expressions and Substitution

0.0 Mathematical expressions and formulas

0.0.0 Conventions for syntactic definitions

We assume familiarity with the elements of formal syntax, as used in introductory programming courses. Here we briefly recapitulate some of them, together with conventions that provide precision with minimal notational clutter.

A common style for syntax definition is a phrase structure grammar, which is a collection of production rules. An important class are the context free grammars (CFGs), where the left-hand side of every production rule is a single nonterminal.

An example is the following CFG specifying the syntax for CFG’s as proposed by Wirth [30] in a reaction to the unnecessary diversity syntax definition styles. The alphabet is the set of terminal symbols, distinguished by quote marks.

(0.0) Definition: A self-defining syntax for CFG’s

```
syntax ::= {production}.
production ::= nonterminal "::=" regular ".".
regular ::= sequence {"|" sequence}.
sequence ::= item {item}.
item ::= nonterminal | terminal | "("regular")" | "{" regular "}" | "[" regular "]".
terminal ::= """"character{character}"""".
```

Wirth points out the following advantages of this formulation.

- It requires no other symbols than the standard ASCII set.
- It clearly distinguishes between metasymbols, terminals and nonterminals.
- Characters used as metasymbols (e.g., \|) can also appear in the language.
- Repetition \{ \} (for 0 or more times) eliminates recursion for simple cases.

Indeed, \{item\} is equivalent to items defined by items ::= \\epsilon \mid item items. Furthermore, [item] is another way of writing \\epsilon \mid item and parentheses are used for grouping, e.g. (a \mid b) c is the same as ac \mid bc. Mere ASCII is undesirable for mathematics, so we replace the quote marks for terminals by an underscore.
The set of sentences produced by a terminal symbol is called a syntactic category. Whenever possible, we adhere to the following convention: we shall use a multiletter word in lowercase for a nonterminal symbol, and the first letter of that word in uppercase for the corresponding syntactic category. These conventions combine conveniently with the custom of designating a set by an uppercase letter and using corresponding lowercase symbols as variables ranging over that set.

For instance, if expression is a nonterminal symbol, then the syntactic category of expressions is written E, and an arbitrary element of E is denoted by e, which is a metavariable. If more metavariables are needed, we use letters that are close in the common alphabet (like d and e) or primes (as in e, d, e′).

Elements of syntactic categories are sequences of symbols. We write them without underscores and by simple juxtaposition but, to emphasize their syntactic (uninterpreted) nature, enclosed in quote marks ‘ ’ (e.g., ‘375’ in the language of numerals and ‘x + y’ in the language of expressions). If we also include metavariables (chosen distinct from terminal symbols), we use quasi-quotes [ ].

### 0.0.1 Syntax of simple mathematical expressions and formulas

In most of mathematics, including arithmetic and logic, expressions consisting of more than one symbol can be seen as applications, namely combinations of a function symbol and an argument. For instance, ‘x + 3’ is the application of the function symbol + to the argument ‘x, 3’ (a pair). In ‘∀ P’, the function symbol is ∀ and the argument is ‘P’. Let us, for the time being, consider only one-argument functions with prefix notation and two-argument functions with infix notation. The syntax of such expressions can be described as follows.

Assume we have at our disposal a set V of symbols designated as variables, a set C₀ of symbols designated as constants and sets C₁ and C₂ of symbols designated as one-place and two-place function symbols. Synonyms for one-place, two-place and function symbol are monadic, dyadic and operator respectively.

(0.1) Example: variables and constants Let us choose V = {a, b, z,}, C₀ = {a, b, c}, C₁ = {succ, pred}, C₂ = {+, −} as a running example.

On the basis of V, C, and the additional symbols ( and ), we now define the set E of expressions (or language E).

<table>
<thead>
<tr>
<th>ref.</th>
<th>form</th>
<th>legend for the metavariables appearing in the form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ev:</td>
<td>[v]</td>
<td>v is a variable from V</td>
</tr>
<tr>
<td>E0:</td>
<td>[c]</td>
<td>c is a constant from C₀</td>
</tr>
<tr>
<td>E1:</td>
<td>[[(φ e)]]</td>
<td>φ is an operator from C₁ and e is an expression</td>
</tr>
<tr>
<td>E2:</td>
<td>[[(d * e)]]</td>
<td>* is an operator from C₂ and d and e are expressions</td>
</tr>
</tbody>
</table>

The same expression syntax can also be defined by a context-free grammar.

```plaintext
expression ::= variable | constant | application
application ::= (cop₁ expression) | (expression cop₂ expression)
```
The forms $[(\phi e)]$ and $[(d \ast e)]$ are called applications of an operator (here designated by the metavariable $\phi$ or $\ast$), to an argument (here designated by the metavariables $d$ and $e$). Observe that, in traditional calculus, the syntax for application of one-place operators is more typically of the form $[\phi(e)]$, as in $f(x) = x^2 + 1$, but we shall see that the convention used here, derived from common practice when working with higher-order functions, ultimately yields considerably less notational clutter.

Definition 0.2 is clearly recursive since, for instance, the metavariable $e$ in an expression of the form $[[(\phi e)]]$ again stands for an expression constructed according to the same definition in which $[[(\phi e)]]$ appears as one of the alternatives.

(0.3) Example Typical expressions using symbols from Example 0.1 are:

\[ 'a' \ 'x' \ '(', \text{succ} \ y \ ')' \ '(', a + y \ ')' \ '(', (a \cdot (\text{succ} \ x)) + y \ ')' \]

Obviously, the syntax of Definition 0.2 causes the appearance of many more parentheses than is convenient for actual practice. For instance, one prefers writing \( 'a \cdot \text{succ} \ x + y' \) rather than \( '((a \cdot (\text{succ} \ x)) + y)' \). Redefining the syntax would spoil the conceptual simplicity of Definition 0.2. Providing separate rules for removing or (re)introducing parentheses is a simpler solution. A requirement is then that expressions in the alternative notation can always be unambiguously restored to the basic syntax\(^0\). Here the rules are the following.

Outer parentheses may always be omitted, e.g. \( 'x + y' \) stands for \( 'x + y' \).

The removal of inner parentheses is governed by the following operator precedence convention. Parentheses around an application of an operator with given precedence may be omitted iff that application occurs as one of the arguments of an infix operator with lower precedence. Precedence is usually assigned in such a way that the number of parentheses can be minimized on the average.

As a general rule, prefix operators receive have higher precedence than infix operators. The mutual precedence of infix operators must be specified explicitly.

(0.4) Example \( '\text{succ} \ x + y' \) stands for \( '(\text{succ} \ x) + y' \) and (hence) not for \( '\text{succ} \ (x + y)' \). If the latter is meant, the parentheses must remain. Assuming that $\cdot$ has higher precedence than $+$, parentheses in \( '(x \cdot y) + z' \) may be omitted to yield \( 'x \cdot y + z' \), but those in \( x \cdot (y + z) \) must remain.

Among the possible one-place and two-place function symbols one might wish to introduce a special class which we call predicate symbols. Expressions that are applications of such predicate symbols\(^1\) are called (atomic) formulas.

(0.5) Example Interesting predicate symbols we could introduce in our running example are the one-place symbol $\text{iszero}$ (zero check), and the two-place symbols $\equiv$ (equality), $\leq$ (less than). Typical formulas are then

\[ '\text{iszero} \ x' \ 'x + a = y' \ '\text{succ} \ x = x + 1' \ 'x < \text{succ} \ x' \]

In general, infix predicate symbols are given lower precedence than any non-predicate symbol, so \( 'x + a = y' \) stands for \( '(x + a) = y' \).

---

\(^0\)The reason is that all properties, semantics, axioms etc. are based on this syntax.

\(^1\)I.e., expressions of the form $[[(\phi e)]]$ or $[[(d \ast e)]]$ where $\phi$ and $\ast$ are predicate symbols.
0.0.2 The meaning of mathematical expressions and formulas

If asked what the equality ‘x + y = y + x’ means, most people would (justifiably) reply: “this means that addition does not depend on the order of the arguments”. However, if asked what ‘x + y’ means, most would (equally justifiably) answer: “that depends on what the values of x and y are”.

Underlying these replies are a number of tacit conventions and assumptions that deserve being made explicit as precisely as possible. The best way for doing this is obviously doing it formally. Hence we shall use mathematical notation as a form of expression at the metalevel, even when such notation is the object of study at the language level. In particular, we shall freely use mathematical functions to map sentences in the language to their meaning.

Defining such meaning functions for our language of expressions is more convenient with the following auxiliary components.

- A domain of interpretation \( D \), which is the set of values of interest.
- A state space \( S := V \to D \), the set of all functions from \( V \) to \( D \). Elements of \( S \) are called states (in some logic textbooks called assignments).
- An interpretation, defined by a family \( k \) of three functions:
  \[
  \begin{align*}
  k_0 & \in C_0 \to D \text{ for the constants,} \\
  k_1 & \in C_1 \to D \to D \text{ for the one-place operators,} \\
  k_2 & \in C_2 \to D^2 \to D \text{ for the two-place operators.}
  \end{align*}
  \]

By convention, \( X \to Y \to Z \) stands for \( X \to (Y \to Z) \), and \( f x y \) for \( (f x) y \). Observe that, for any prefix operator \( \phi : C_1 \), the interpretation \( k_1 \phi \) is a function from \( D \) to \( D \) and that the image \( k_1 \phi d \) of any \( d : D \) is a value in \( D \).

(0.6) Example For arithmetic on natural numbers, the domain of interpretation is \( \mathbb{N} \). The function \( k_0 \) is rather uninteresting since it can be any convention specified in the context (e.g. \( k_0 \underline{\alpha} = 2 \) for the sequel). The function \( k_1 \) maps \( \text{succ} \) to the function \( n : \mathbb{N} \to n + 1 \) or, expressed without an abstraction, \( k_1 \text{succ} n = n + 1 \). The function \( k_2 \) maps \( + \) to the arithmetic sum function, viz. \( k_2 + (m, n) = m + n \) for any \( m \) and \( n \) in \( \mathbb{N} \).

These auxiliary components are assembled in the following definition.

(0.7) Definition: Semantics of Expressions The meaning function

\[
\mathcal{E} : E \to S \to D
\]

for expressions in \( E \) is recursively defined by specifying the meaning of every syntactic construct in Definition 0.2 on page 2 by a separate rule:

<table>
<thead>
<tr>
<th>ref.</th>
<th>image definition for ( \mathcal{E} )</th>
<th>for any state ( s ) in ( S ) and any</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mv:</td>
<td>( \mathcal{E} [v] \ s = s v )</td>
<td>variable ( v ) in ( V )</td>
</tr>
<tr>
<td>M0:</td>
<td>( \mathcal{E} [c] \ s = k_0 c )</td>
<td>constant ( c ) in ( C_0 )</td>
</tr>
<tr>
<td>M1:</td>
<td>( \mathcal{E} [(\phi e)] \ s = k_1 \phi (\mathcal{E} e) \ s )</td>
<td>( \phi ) in ( C_1 ) and ( e ) in ( E )</td>
</tr>
<tr>
<td>M2:</td>
<td>( \mathcal{E} [(d * e)] \ s = k_2 * (\mathcal{E} d \ s, \mathcal{E} e) \ s )</td>
<td>in ( C_2 ), ( d ) in ( E ) and ( e ) in ( E )</td>
</tr>
</tbody>
</table>
(0.8) Example  Let us evaluate the meaning of \( a \cdot \text{succ} \ x + y \). Observe also the format for chaining calculation steps, with 'hints' between \( \langle \rangle \).

\[
\mathcal{E}^{a \cdot \text{succ} \ x + y}s
= \langle \text{Normalize} \rangle \ \mathcal{E}^{(a \cdot (\text{succ} \ x)) + y}s
= \langle \text{Rule M2} \rangle \ k_2 + \langle \mathcal{E}^{(a \cdot (\text{succ} \ x))}'s, \mathcal{E}^{y}'s \rangle
= \langle \text{Rule M2} \rangle \ k_2 + (k_2 \cdot \langle \mathcal{E}^{a'}s, k_1 \text{succ}(\mathcal{E}^{x'}s) \rangle, \mathcal{E}^{y'}s) \rangle
= \langle \text{Rule M0} \rangle \ k_2 + (k_2 \cdot (k_0 \ a, k_1 \text{succ}(s \ x)), \mathcal{E}^{y'}s) \rangle
= \langle \text{Defin. } k_i \rangle \ k_0 \ a \cdot (s \ x + 1) + s \ y
\]

This shows how the recursive definition of \( \mathcal{E} \) distributes the state function \( s \) over the expression. For the particular case where \( s \) satisfies \( s \ x = 3 \) and \( s \ y = 7 \) and assuming the interpretation from Example 0.6, we obtain

\[
\mathcal{E}^{a \cdot \text{succ} \ x + y}s = 2 \cdot (3 + 1) + 7.
\]

Predicate symbols are interpreted as functions that can take only the value 0 and 1. Reasons for representing truth values by numbers rather than letters like \( F \) and \( T \) will become increasingly apparent as we proceed. We write \( \mathbb{B} \) for the set \( \{0, 1\} \) of booleans. Later on we will (very briefly) talk about fuzzy predicates that can take values in the closed interval \([0, 1] \). Hence the meaning of a formula is a function mapping the state to either 0 or 1.

(0.9) Examples  \( \mathcal{E}^{x + a = y} s = 1 \) for any state \( s \) with \( s \ x + k_0 \ a = s \ y \) and 0 otherwise. Similarly, \( \mathcal{E}^{\text{succ} \ x = x + 1} s = 1 \) for any state \( s \).

We say that a collection \( F\text{mls} \) of formulas semantically implies a formula \( p \), written \( F\text{mls} \models p \), if \( \mathcal{E} p s = 1 \) for every state \( s \) in which the meaning of every formula in \( F\text{mls} \) evaluates to 1. Equivalently, the set of states satisfying \( F\text{mls} \) contains the set of states satisfying \( F\text{mls} \). If \( p \) has this property for empty \( F\text{mls} \), then \( \mathcal{E} p s = 1 \) irrespective of \( s \), and \( p \) is called (semantically) valid, written \( \models p \). The presence of \( \models \) makes the (quasi-)quotes around the formulas optional.

(0.10) Examples  \( x > y \models \text{succ} \ x > \text{succ} \ y \)  \( \models \text{succ} \ x > x \)

The standard convention in everyday mathematics is that, if a theorem is expressed as a formula, this formula is implicitly understood to be valid, i.e. its meaning evaluates to 1 for all states. An example is \( x + y = y + x \), expressing the commutativity of addition.

0.1 Substitution and equality

Unless explicitly stated otherwise, we put all interpretation and meaning aside, and again consider expressions as purely syntactic entities.
0.1.0 Substitution in mathematical expressions and formulas

One of the most important syntactic operations on expressions is substitution. For instance, substituting ‘z · a’ for \( x \) in ‘\( x + y \)’ yields ‘\( z · a + y \)’.

This informal example is also meant to issue a warning: one must imagine omitted parentheses to be present in their proper places, i.e., the preceding formulation is shorthand for “substituting ‘(z · a)’ for \( x \) in ‘(x + y)’ yields ‘((z · a) + y)’”. The danger inherent in sloppiness is illustrated by interchanging the ± and · signs.

Indeed, substituting ‘\( z + a \)’ for \( x \) in ‘\( x · y \)’ yields the correct result ‘\( (z + a) · y \)’ if the warning is heeded, but the erroneous result ‘\( z + a · y \)’ if one forgets that parentheses around \( z + a \) are optional only in case ‘\( z + a \)’ stands by itself.

It is customary to write \( e[w := g] \) for the result of substituting the expression \( g \) for the variable \( x \) in the expression \( e \), for instance, \( (x + y)[x := z · a] = (z · a) + y \).

Hence \( [w := g] \) can be seen as an expression transformer, since it maps the expression \( e \) to the expression \( e[w := g] \). Since the argument is clearly syntactic, enclosing it in \( [ \] \) or ‘ \( \) ’ is optional (the exact conventions are not given here).

As the reader will expect by now, the formal definition of substitution in expressions is again recursive on the syntactic structure of those expressions. The conditional expression \( b ? x y \) (read “if \( b \) then \( x \) else \( y \)”) is self-explanatory.

<table>
<thead>
<tr>
<th>(0.11) Definition: Formal Substitution</th>
<th>For any given variable ( w ) and expression ( g ), we define the postfix operator ([w := g]) as follows.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref.</td>
<td>image definition for ([w := g])</td>
</tr>
<tr>
<td>( \text{Sv:} )</td>
<td>( v[w := g] = (v = w) ? g ) ( v ) ( \downarrow [v] )</td>
</tr>
<tr>
<td>( \text{S0:} )</td>
<td>( c[w := g] = [c] )</td>
</tr>
<tr>
<td>( \text{S1:} )</td>
<td>( (\phi e)[w := g] = [{(\phi e[w := g])}] )</td>
</tr>
<tr>
<td>( \text{S2:} )</td>
<td>( (d * e)[w := g] = [{(d[w := g] * e[w := g])}] )</td>
</tr>
</tbody>
</table>

We illustrate this by means of an example, which shows how the recursive definition distributes the expression transformer over the components of the argument.

\((a \cdot \text{succ} x + y)[x := z \cdot b]\)

\[
\begin{align*}
(a \cdot \text{succ} x + y)[x := z \cdot b] & = \text{(Normalize) } '((a \cdot \text{succ} x) + y)[x := (z \cdot b)] \\
& = \text{(Rule S2) } \left[((a \cdot \text{succ} x)[x := (z \cdot b)] + y[x := (z \cdot b)])\right] \\
& = \text{(Rule S2) } \left[((a[x := (z \cdot b)] \cdot \text{succ} x)[x := (z \cdot b)] + y[x := (z \cdot b)])\right] \\
& = \text{(Rule S1) } \left[((a[x := (z \cdot b)] \cdot \text{succ} x)[x := (z \cdot b)] + y[x := (z \cdot b)])\right] \\
& = \text{(Rule S0) } \left[((a \cdot \text{succ} x[x := (z \cdot b)]) + y[x := (z \cdot b)])\right] \\
& = \text{(Rule SV) } '((a \cdot \text{succ} (z \cdot b)) + y) \\
& = \text{(Opt. par.) } 'a \cdot \text{succ} (z \cdot b) + y'
\end{align*}
\]

(0.13) Convention To save space, we often write \( e[w := g] \) as \( e^w \).

The notation \([w := g]\) is generalized in a self-evident way to multiple substitutions, where \( w \) is a list of variables and \( g \) a list of expressions of equal length.
0.1. *Substitution and equality*

(0.14) **Example** \( (x + y)[x, y := z \cdot a, y \cdot b] = 'z \cdot a + y \cdot b'\)

Note that the substitutions are to be made in parallel, not consecutively.

(0.15) **Example** By multiple substitution, \( (x + y)[x, y := y, x] = 'y + x'\).

However, by consecutive substitution, \( (x + y)[x := y][y := x] = 'x + x'\).

0.1.1 **Instantiation of theorems as a strict inference rule**

In general, an *inference rule* is a little table of the form

\[
\begin{array}{c}
\mathcal{P} \\
\hline
q
\end{array}
\]

where \( \mathcal{P} \) is a collection of formulas called *premises* and \( q \) is a formula called the *conclusion* or *direct consequence*. All items in the table are purely *syntactic*. However, the *semantic* significance is clear from the *soundness* condition, namely that the inference rule must guarantee that \( \mathcal{P} \models q \). Verifying whether this condition is satisfied can be done by induction on the structure of \( q \).

An important special case is when all of the premises in \( \mathcal{P} \) are *theorems*. How we obtain theorems *syntactically* in the first place is discussed in Section 0.1.3 on page 9. In terms of *semantics*, a formula is a theorem iff it is valid, i.e., its meaning evaluates to 1 in every state. Hence, if the premises of a (sound) inference rule are theorems, then so is the conclusion. If we *require* that all the premises are theorems, the inference rule is called *strict*. In that case, we need establish soundness only for premises that are theorems.

An example of a strict inference rule is *instantiation of theorems*, as justified (informally) next. Indeed, assume we have a formula that is a theorem. Hence, if the value (assigned by the state) of a variable is modified, then the meaning of that formula still evaluates to 1. As a *syntactic* consequence, that variable can as well be replaced by any expression, provided this is done uniformly, viz. if all occurrences of that variable are replaced by the same expression. Substitution precisely accomplishes such a uniform replacement. These observations justify the following inference rule. It has a single premiss, which must be a theorem.

\[
(0.16) \text{INFEERENCE RULE: INSTANTIATION (OF A THEOREM)} \quad \frac{p}{p[v := e]}
\]

for any theorem \( p \), (list of) variable(s) \( v \) and (list of) expression(s) \( e \).

The theorem \( p[v := e] \) is called an *instance* of theorem \( p \).

(0.17) **Example** Given that the formula \( x + y = y + x \) is a theorem, then \( (x + y = y + x)[x, y := z \cdot a, y \cdot b] \) or, equivalently, \( z \cdot a + y \cdot b = y \cdot b + z \cdot a \), is also a theorem by instantiation.

0.1.2 **Equality and Leibniz’s principle**

In mathematics, entities (objects, \ldots) are considered *equal* iff they are indistinguishable for *all* intents and purposes. This can be expressed mathematically in
the following semi-formal way. Let $P$ be a property or test pertaining to mathematical entities, and let us write $P e$ for “entity $e$ has property $P$”. Then we say that $e = e'$ iff, for any such property $P$ that we can devise, $P e$ holds iff $P e'$ holds. If we had quantification available at this stage, we could express this formally

$$e = e' \iff \forall P : \text{Prop} . (P e \equiv P e')$$

where $\equiv$ stands for “if and only if”. We shall not further belabor this point.

In high school algebra, one learns that equality has the following properties.

- Reflexivity: $e = e$
- Symmetry: given $e = e'$ then $e' = e$
- Transitivity: given $e = e'$ and $e' = e''$ then $e = e''$

However, equality entails considerably more. Indeed, any equivalence relation already satisfies these three properties by definition. Equivalence means indistinguishability only for some (specific) properties, intents or purposes.

(0.18) Example In arithmetic, equivalence modulo $n$, written $\equiv_n$, is defined by $x \equiv_n y$ iff $n \mid (x - y)$. Here $n \mid m$ stands for “$n$ divides $m$”. Hence equivalence modulo $n$ means equality up to a multiple of $n$. Reflexivity, symmetry and transitivity are easily verified using high school algebra.

Equality means indistinguishability w.r.t. all properties and purposes, except, of course, w.r.t. the literal text of the expressions themselves (which may differ).

Syntactically, this additional property is reflected by the fact that, if $e = e'$, then any appearance of $e$ may be replaced by $e'$ under any circumstances. This is the informal statement of Leibniz’s principle. Substitution differs from replacement in that it pertains only to variables (not larger expressions) and that substitution replaces all occurrences of the specified variables.

Yet, substitution as formalized by the expression transformer allows expressing replacement precisely: it suffices to use a new variable (i.e., one that is not used elsewhere), as a placeholder for the subexpressions to be replaced.

(0.19) Example The equality obtained from the simple calculation step

$$ (a^2 - b^2) + b^2 = \langle a^2 - b^2 = (a + b) \cdot (a - b) \rangle \quad (a + b) \cdot (a - b) + b^2 $$

can be written as follows, using $z$ as a placeholder for the replaced part.

$$ (z + b^2)[z := a^2 - b^2] = \langle (z + b^2)[z := (a + b) \cdot (a - b)] \rangle $$

This placeholder technique is used only for the purpose of formalizing Leibniz’s principle: in actual calculation no new variables are introduced.

Generalized, this pattern expresses Leibniz’s principle as an inference rule.

(0.20) Inference Rule: Leibniz’s Principle

For any expressions $e$, $d'$, $d''$ in $E$ and any variable $v$ in $V$,

$$ d' = d'' \quad \Rightarrow \quad e[v := d'] = e[v := d''].$$


0.1.3 Formal calculation with instantiation and equality

Here we present the absolutely minimal framework for doing certain simple kinds of proofs, assuming propositional and predicate calculus are not yet available. However, the basic pattern readily supports later extensions.

(0.21) Definition: Deduction Given a collection of formulas, called hypotheses, then a consequence of those hypotheses is a formula that is

- either one of the hypotheses themselves,
- or the conclusion of an inference rule whose premises are consequences of the hypotheses.

A deduction from a set of hypotheses to a consequence is the collection of (intermediate) consequences used in establishing that consequence, organized in some systematic way: a directed graph, a tree, a sequence of formulas with one-directional references, or —indeed— a calculation.

We write $\mathcal{H} \vdash q$ in case a formula $q$ is a consequence of the collection $\mathcal{H}$ of hypotheses. If all the inference rules used are sound, then $\mathcal{H} \vdash q$ is a sufficient condition for $\mathcal{H} \models q$.

Axioms are hypotheses —hence purely syntactic formulas— but chosen in such a way that they are (semantically) valid under some relevant interpretation. Consequences of the axioms are called theorems. This relationship is illustrated pictorially in Fig. 0.0.

![Figure 0.0: Cranking out theorems](image)

Here we consider only simple theories in which the theorems are equalities, like the arithmetic equalities we have seen in the examples thus far.

The inference rules are instantiation and the laws for equality.

0. Instantiation (strict):

\[ p \quad \frac{p[v := e]}{p[v := e]} \quad (\alpha) \]

1. Leibniz’s principle (non-strict):

\[ d = d' \quad e[v := d] = e[v := d'] \quad (\beta) \]

2. Symmetry of equality (non-strict):

\[ e = e' \quad e' = e \quad (\gamma) \]

3. Transitivity of equality (non-strict):

\[ e = e' , e' = e'' \quad e = e'' \quad (\delta) \]
In the calculational style, deductions are written in the general format

\[(0.22) \quad e_0 = \langle \text{justification}_0 \rangle \quad e_1 = \langle \text{justification}_1 \rangle \quad e_1 \quad \text{and so on.}\]

The inference rules for equality are embedded uniformly in this format as follows. The use of an inference rule with one premiss \( p \) and conclusion \( e' = e'' \) is written

\[(0.23) \quad e' = \langle p \rangle \quad e''.\]

We briefly elaborate and exemplify this for each of the inference rules, taking into account that (a) in equational calculation, all formulas (including premisses) are equalities, and for instantiation even theorems; (b) meta-expressions of the form \( e[v := d] \) are used just for exposition at the meta-level: the calculational format shows the expressions only in their final form after substitution.

0. **Instantiation** The premiss \( p \) is a theorem of the form \( d' = d'' \), and hence the conclusion \( p[v := e] \) is the theorem \( d'[v := e] = d''[v := e] \) which has the form \( e' = e'' \). Example: \( x \cdot (a + y) = (x \cdot y + y \cdot x) \cdot (a + y) \cdot x \).

Clariﬁcation: in \((\alpha)\), write \( x \cdot y = y \cdot x \) for \( p \), \( a + y \) for \( e \) and \( y \) for \( v \). The fact that \( 5 = (x = 3 \cdot x) \cdot 3 \cdot 5 \) is “incorrect” illustrates strictness.

1. **Leibniz** The premiss \( p \), not necessarily a theorem, is of the form \( d' = d'' \) and the conclusion \( e[v := d'] = e[v := d''] \) is of the form \( e' = e'' \). Example: if \( y = a \cdot x \), then we may write \( x + y = (y = a \cdot x) \cdot x + a \cdot x \).

Clariﬁcation: in \((\beta)\), write \( y \) for \( d' \), \( a \cdot x \) for \( d'' \), \( x + z \) for \( e \) and \( y \) for \( v \). The fact that \( y = a \cdot x \) is just a hypothesis, not a theorem, illustrates non-strictness.

2. **Symmetry** The premiss \( p \), not necessarily a theorem, is of the form \( e'' = e' \).

However, this simple step is usually taken tacitly, as explained below.

3. **Transitivity**, which has two equalities for premisses, is used implicitly to justify chaining \( e_0 = e_1 \) and \( e_1 = e_2 \) in the general format \((0.22)\) to conclude that \( e_0 = e_2 \) etc.. Observe how this avoids writing every expression twice.

Conventions for making calculations even more succinct are the following.

Tacit instantiation when using Leibniz’s principle: if the premiss \( d' = d'' \) in \((\beta)\) is the instantiation of some theorem \( p \), then we still write the deduction step as in \((0.23)\), leaving the instantiation from \( p \) to \( d' = d'' \) implicit. Example: \( x \cdot z \cdot (a + y) = (x \cdot y = y \cdot x) \cdot x + (a + y) \cdot z \).

Only when this instantiation is not evident shall we write the justification as \( \langle p[v := e] \rangle \) or \( \langle p \text{ where } v := e \rangle \).

Tacit use of symmetry when using Leibniz’s principle: if the sides of the equality in the premiss of \((\beta)\) are not in the desired order, they are swapped tacitly, as expressed in the metatheorem \( d' = d'' \implies e[v := d'] = d[v := d''] \).

Example: tacitly swapping \( y = a \cdot x \) to \( a \cdot x = y \) in \( x + a \cdot x = (y = a \cdot x) \cdot x + y \).

Tacit use of symmetry in the entire calculation, viz., deducing \( e_k = e_0 \) to deduce \( e_0 = e_k \). Such reversal is helpful if the chain from \( e_k \) to \( e_0 \) is easier. This is ofen the case when \( e_k \) is “more structured” in some sense than \( e_0 \). For instance, proving \( x^2 - y^2 = (x + y) \cdot (x - y) \) is easier starting from the right (why?).
0.2 Elements of the lambda calculus

In Chapter 3 we shall see that formulas like $\forall x: \mathbb{R} . x^2 \geq 0$ consist of a quantifier $\forall$ and a predicate written in Funmath abstractor notation as $x: \mathbb{R} . x^2 \geq 0$. The general form of such an abstraction is $v : S \land p . e$, which denotes a function whose mapping is defined for appropriate $d$ (specified later) by $(v : S \land p . e) d = e[v := d]$.

Clearly, substitution plays an important role. So here is the proper place to discuss the underlying principles and calculation rules, provided we ignore later refinements such as function domains, types, and domain-specific (arithmetic, logic, ...) operators. What remains is the so-called pure lambda calculus. Note that, although one usually talks about ‘the’ lambda calculus, there are many variants. A more complete treatment can be found in a textbook by Barendregt [2].

We avoid discussing semantics, and calculate formally with expressions, called lambda terms, with rules for equality and 3 conversion rules, no other axioms. Yet, we show how suitable lambda terms can be used as representations for mathematical entities (numbers, arithmetic functions). Although lambda calculus exists since the 1930’s, the first satisfactory models were devised only ca. 1970 by Dana Scott using lattice theory (which is beyond the scope of these notes).

0.2.0 Syntax, nomenclature and conventions

The expressions are the (lambda) terms defined by the following CFG.

\[
\text{term} ::= \text{variable} \mid (\text{term} \text{ term}) \mid (\lambda \text{variable}. \text{term})
\]

We write $\Lambda$ for the syntactical category of terms. We use $L$, $M$, $N$, $P$, $Q$, $R$ as metavariables for arbitrary terms in $\Lambda$ and $u,v,w$ as metavariables for arbitrary variables in $V$. In the examples, we assume $V := \{xyzpq\}$. We will also introduce symbols like $\textbf{C, D, I, K, T, Y}$ as abbreviations for certain terms. Since the entire discussion is purely syntactic, we can dispense with all kinds of quotes.

\[
(0.25) \text{Examples } x \quad (xy) \quad (\lambda x.x) \quad (\lambda x.(\lambda y.(x(yz))))
\]

A term of the form $(MN)$ is called an application and $(\lambda x.M)$ is an abstraction. In $(\lambda x.M)$, the part $\lambda x.$ is called the abstractor and $M$ the abstraction or body.

Certain parentheses are made optional by the following conventions.

0. The outer parentheses in terms of the form $(MN)$ or $(\lambda x.M)$ are optional if this term stands by itself or as an abstraction. The latter implies that the scope of an abstractor extends to the right as far as possible.

1. Application associates to the left, e.g., $(LMN)$ stands for $((LM).N)$.

   Warning: reading $(LMN)$ as $(L(MN))$ is the main source of errors.

\[
(0.26) \text{Examples } \text{ By rule 0, } \lambda x.(\lambda y.xy)x \text{ stands for } (\lambda x.(\lambda y.(xy))x).
\]

By rule 1, $(MNQP)$ stands for $(((MN)P)Q)$. Warning: in $(L(MN))$, the inner parentheses must remain, and $(M(NPQ))$ stands for $(M((NP)Q))$.

Nested abstractions may be merged by writing $\lambda u.\lambda v.M$ as $\lambda uv.M$.

Together, these conventions allow writing, e.g., $\lambda x.y(\lambda xy.xz(\lambda z.(yz)))yz$ for $(\lambda x.((y(\lambda x.((xz)(\lambda z.(xy))))))y)z)$, which saves 18 parentheses!
0.2.1 Calculation rules: substitution, equality and conversion

In $\lambda v. M$, the scope of $\lambda v$ is $M$. Every occurrence of $v$ in $\lambda v. M$ is called bound. Occurrences that are not bound are called free. We recursively define an operator $\varphi$ such that the set of variables with free occurrences in a term $L$ is $\varphi L$:

$$\varphi [v] = \{v\} \quad \varphi [(MN)] = \varphi M \cup \varphi N \quad \varphi [(\lambda v. M)] = (\varphi M) \setminus \{v\}.$$

Recall that $\{v\}$ denotes the singleton set containing just $v$. The set difference $A \setminus B$ for any given sets $A$ and $B$ is the set of elements in $A$ that are not in $B$. A closed term or (lambda-)combinator is a (lambda) term without free variables.

(0.27) Example Let the variable occurrences in $\lambda x.y(\lambda x.y.xz(\lambda z.xy)y)yz$ be numbered from 0 to 11, then the only free occurrences are those of $y$ and $z$ in positions 1, 5, 10 and 11. Exercise: derive $\varphi '\lambda x.y.xz'y = \{x, y\}$.

Using convention 0.13 (page 6), the substitution $[w]_L$ with variable $w$ and term $L$ is defined recursively as follows. For the conditional expression, $(c ? e_1 \dagger e_0) = e_c$.

<table>
<thead>
<tr>
<th>Svar:</th>
<th>$v[w]_L = (v = w)? L \downarrow [v]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sapp:</td>
<td>$(MN)[w]_L = (M[y]_N)\downarrow [w]$</td>
</tr>
<tr>
<td>Sabs:</td>
<td>$(\lambda u. M)[w]_L = (v = w)? (\lambda v. M) \downarrow (v \notin \varphi L)? (\lambda v. M[y]_N) \downarrow (\lambda u. M[w]_N)_L$</td>
</tr>
</tbody>
</table>

The calculation rules for the lambda calculus are the following.

a. The rules of equality: symmetry, transitivity and Leibniz’s principle:

$$M = N \quad L = M \quad M = N \quad L[v := M] = M[v := N].$$

Taking for $L$ in Leibniz’s rule the particular forms $L, vL$ and $Lv$ (assuming $v \notin \varphi L$) yields reflexivity, left monotonicity and right monotonicity.

b. The proper axioms common to most variants of the lambda calculus

(0.28) Axioms of the lambda calculus

- β-conversion: $(\lambda v. M)N = M_N^v$
- α-conversion: $(\lambda v. M) = (\lambda w. M[w^v])$ provided $w \notin \varphi M$

Certain authors [2] consider α-conversion subsumed by syntactic equality.

c. Specific additional axioms characterizing variants of the lambda calculus.

(i) Rule ξ: $M = N \quad \lambda v. M = \lambda v. N$  
(ii) Rule η (also called η-conversion): $(\lambda v. Mv) = M$ provided $v \notin \varphi M$
(iii) Rule ζ (also called extensionality): $Mv = Nv \quad \text{provided } v \notin \varphi (M, N)$

It is easy to show (exercise) that, as an addition to the rules in (a) and (b), rule ζ (extensionality) is equivalent to rules ξ and η combined. In the sequel, we assume extensionality, hence “everything”.

---

12 Chapter 0. Expressions and Substitution
\[ (\lambda xy. xy)yx = (\beta\text{-conversion}) \quad ((\lambda y. xy)z)_y^x x \]
\[ = (\text{Rule Sabs}) \quad (\lambda z. (xy)_y^x)z_x^y x \]
\[ = (\text{Rule Sapp}) \quad (\lambda z. (xz)_y^x)z_x^y x \]
\[ = (\text{Rule Svar}) \quad (\lambda z. yz)z_x^y x \]
\[ = (\beta\text{-conversion}) \quad (yz)_x^y z_x^y \]
\[ = (\text{Rule Sapp}) \quad y((x)_y^x) z_x^y \]
\[ = (\text{Rule Svar}) \quad yx \]

Observe that, if the rule Sabs did not take precaution against name clashes, then \((\lambda y. xy)z)_y^x x\) would evaluate to \((\lambda y. yy)z_\) and the complete expression \((\lambda xy. xy)yx\) would evaluate to the erroneous result \(xx\).

0.2.2 Shortcuts and their application to lambda combinators

Terms that can be converted into each other by \(\alpha\)-conversion are \(\alpha\)-congruent. Since \(\alpha\)-conversion is included in the axioms in all variants of interest, we make it implicit henceforth considering \(\alpha\)-congruent terms to be indistinguishable.

In a finite context, substitution in abstractions is reduced to its naive form
\[
(\lambda v. M)_v^w = \lambda v. M \]
by assuming that the dummies in an abstraction are chosen different from each other (eliminating the case \(v = w\)) and from variables that occur free in any other term in the same context (eliminating the case \(v \in \varphi L\) or, more directly, that all dummies are new variables. A simple way to achieve this in hand calculations is tacitly applying alpha conversion (if needed) before the calculation.

Most interesting calculations involve closed terms or (lambda-)combinators, namely lambda terms containing no free variables.

\[ (\lambda uv. uv)MN = (\beta\text{-conversion}) \quad ((\lambda uv. uv)M)_M^N \]
\[ = (\text{Naive Sabs}) \quad ((\lambda v. uv)_M^N)N \]
\[ = (\text{Sapp, Svar}) \quad (\lambda v. Mv)_N \]
\[ = (\beta\text{-conversion}) \quad (Mv)_N \]
\[ = (\text{Sapp, Svar}) \quad MN \]

We present some lambda combinators whose names already suggest their purpose, but whose interesting properties will gradually become clear.
(0.31) Definitions: Some useful lambda combinators

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Designated term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition</td>
<td>C</td>
<td>( \lambda yz.x(yz) )</td>
</tr>
<tr>
<td>Duplication</td>
<td>D</td>
<td>( \lambda x.xx )</td>
</tr>
<tr>
<td>Identity</td>
<td>I</td>
<td>( \lambda x.x )</td>
</tr>
<tr>
<td>Constant</td>
<td>K</td>
<td>( \lambda xy.y )</td>
</tr>
<tr>
<td>Shuffle</td>
<td>S</td>
<td>( \lambda xz.xz(yz) )</td>
</tr>
<tr>
<td>Reproducing</td>
<td>R</td>
<td>( (\lambda x.xx)(\lambda x.xx) )</td>
</tr>
<tr>
<td>Transposition</td>
<td>T</td>
<td>( \lambda xyz.zxy )</td>
</tr>
<tr>
<td>Paradoxical</td>
<td>Y</td>
<td>( \lambda x.(\lambda y.x(yy))(\lambda y.x(yy)) )</td>
</tr>
</tbody>
</table>

In principle, the symbols are meant as textual abbreviations for the corresponding terms in the sense that, for instance \( \lambda xy.Kyx \) stands for \( \lambda xy.(\lambda xy.y)yx \). The resulting expressions usually do not satisfy the conditions for the short cuts, e.g., in converting the abstrahend in \( \lambda xy.(\lambda xy.x)yx \), which should reduce the entire expression to \( \lambda xy.y \). However, since all variables in a lambda combinator are bound, \( \alpha \)-conversion allows to replace them by new variables, thereby satisfying the conditions for the short cuts. We can even assume that this is done when the symbol is replaced by the expression, e.g., \( \lambda xy.Kyx \) then stands for \( \lambda xy.(\lambda pq.p)yx \).

Now the shortcut can be carried further: applying \( \beta \)-conversion to an application of an abstraction to a term often yields a term with fewer abstractions than the application, for instance \( CN = \lambda yz.N(yz) \). For some lambda combinators, even all original abstractors can be eliminated, as shown in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition</td>
<td>C</td>
<td>( CLMN = L(MN) )</td>
</tr>
<tr>
<td>Duplication</td>
<td>D</td>
<td>( DM = MM )</td>
</tr>
<tr>
<td>Identity</td>
<td>I</td>
<td>( IN = N )</td>
</tr>
<tr>
<td>Constant</td>
<td>K</td>
<td>( KMN = M )</td>
</tr>
<tr>
<td>Shuffle</td>
<td>S</td>
<td>( SLMN = LN(MN) )</td>
</tr>
<tr>
<td>Transposition</td>
<td>T</td>
<td>( TLMN = LNM )</td>
</tr>
</tbody>
</table>

Since the stated properties hold for arbitrary terms \( L, M \) and \( N \), they can be used directly as calculation rules, as in

\[
\lambda xy.Kyx = \langle \lambda MN = M \rangle \lambda xy.y
\]

The reader should attempt a few exercises without and with shortcuts, to appreciate the difference. Henceforth, we will use the shortcuts most of the time.

Observe that \( R \) and \( Y \) behave differently, and even in a way that prevents their inclusion in the second table. In fact, since \( R \) is an application with an abstraction in the operator position, we can apply \( \beta \)-conversion, but the result is again \( R \). Such issues are addressed next.

### 0.2.3 Redexes and normal forms

For further discussion, some additional nomenclature is useful.
0.2. Elements of the lambda calculus

(0.32) Definitions: redexes and normal forms
A $\beta$-redex is an application with an abstraction in the operator position. An $\eta$-redex is an abstraction of the form $\lambda v. M v$ where $v \notin \varphi M$.

A $\beta$-normal form is a term in which no subterm is a $\beta$-redex. A $\beta\eta$-normal form is a term in which no subterm is a $\beta$- or an $\eta$-redex. Usually, “normal form” is abbreviated to “nf”.

Letting $r$ stand for either $\beta$ or $\beta\eta$, we say that a term has an $r$-normal form if there is an $r$-normal form that is equal to it under the $r$-conversion rules. By normal form (without qualification), we mean $\beta\eta$-normal form.

By definition, a $\beta$-redex has the general form $(\lambda v. M) N$. The name indicates that it can be “reduced” to $M \upharpoonright^\lfloor N \rfloor$, although the result need not be a “shorter” term. If $\lambda v. M v$ is an $\eta$-redex, i.e., if $v \notin \varphi M$, it reduces to $M$ (which is shorter).

(0.33) Examples The following terms are $\beta$-redexes.

$$(\lambda x y. x) z \quad (\lambda x y. x)(\lambda x. x) \quad (\lambda x. x)(\lambda x. x)$$

The following terms are $\eta$-redexes.

$$\lambda y. x y \quad \lambda y. (\lambda x. x)(\lambda x. x)y$$

The following terms are in $\beta$-nf.

$$\lambda y. z \quad \lambda y. x \quad x(\lambda y. y)z$$

This is perhaps a good place to repeat our earlier warning: $x(\lambda y. y)z$ does not contain a $\beta$-redex, since it stands for $(x(\lambda y. y)) z$.

More generally, students are often tempted to parse an application of the form $L(\lambda v. M) N$ incorrectly as $L(\lambda v. M) N$, followed by $\beta$-reduction to obtain $LM \upharpoonright^\lfloor N \rfloor$. Recall that $L(\lambda v. M) N$ stands for $(L(\lambda v. M)) N$!

Examples of terms that have no nf are $R$ and $Y$, viz., $(\lambda x. x)(\lambda x. x)$ and $\lambda x. (\lambda y. x(yy))(\lambda y. x(yy))$.

We conclude this brief discussion on nf’s by mentioning two important properties.

- A term has a $\beta$-nf if and only if it has a $\beta\eta$-nf.
- A term has at most one nf (considering $\alpha$-congruent terms to be equal).

The proof of these properties is beyond the scope of this first introduction to the lambda-calculus. It is worthwhile to mention, however, that the usual proofs rely on the notion of reduction, which is a directional variant of conversion, viz.,

$$(\lambda v. M) N \rightarrow_\beta M[\upsilon := N] \quad \text{and} \quad \lambda v. M \upsilon \rightarrow_\eta M, \upsilon \notin \varphi M$$

We will briefly come back to this in the chapter on relations.
0.3 Variants of the lambda calculus

0.3.0 Combinator calculus

The syntax of the pure lambda calculus contained variables but no constants. A variant is combinator theory which, in its pure form, contains only constants but no variables and (hence) no abstractions. Its surprising simplicity is evident in the following definition.

(0.34) Definition: Syntax and calculation rules
The CFG for combinator terms is

\[
\text{term ::= } K \mid S \mid (\text{term term})
\]

The conventions for omitting parentheses are: outer parentheses are optional, and application associates to the left, e.g., \( LMN \) stands for \((LM)N\).

The calculation rules are the rules for equality and the axioms:

\[
KPQ = P \quad \text{and} \quad SPQR = PR(QR)
\]

Together with extensionality, viz., if combinator terms \( M \) and \( N \) satisfy \( ML = NL \) for any combinator term \( L \), then \( M = N \).

Since the language does not contain variables, Leibniz’s principle must be characterized in terms of metavariables, by recursion on the structure of application:

\[
\frac{M = N}{LM = LN} \quad \text{and} \quad \frac{M = N}{ML = NL}
\]

A simple calculation example is the following.

(0.35) Example: calculating with combinators
Let \( M \) and \( N \) be arbitrary combinator terms, then

\[
SKMN = (SPQR = PR(QR)) \quad KN(MN)
= (KPQ = P) \quad N
\]

The example shows that \( SKM \) behaves as an identity operator for any choice of \( M \), in particular if \( M = K \), so we introduce the abbreviation \( I := SKK \).

Combinator terms can be directly transformed into lambda combinators with the same formal properties: it suffices substituting \( K \) for \( K \) and \( S \) for \( S \). It is also the case, but less obvious, that every lambda combinator can be transformed into an equivalent combinator term by eliminating all abstractions.

Doing so is considerably facilitated by allowing variables in combinator terms. At the same time, this makes it possible to transform any lambda term into a combinator term, in which the variables are precisely the free variables in the given lambda term.

Henceforth we assume that the syntax of combinator terms is extended with variables. Moreover, since the intermediate results in the transformation may include abstractions that have not yet been eliminated, we assume the syntax of the pure lambda calculus extended with the constants \( K, S \) and \( I \). We call this system the CL\( \lambda \)-calculus.
0.3. Variants of the lambda calculus

(0.36) Definition: The abstraction eliminator. Given a variable $w$, we define the syntactic operator $\hat{w}$ on $\lambda$-terms recursively as follows.

For variable $w$: \[ \hat{w}w = I \] (Rule I)
For variable $v \neq w$: \[ \hat{w}v = Kv \] (Rule K')
For constant $c$: \[ \hat{w}c = Kc \] (Rule K'')
For an application: \[ \hat{w}(MN) = S(\hat{w}M)(\hat{w}N) \] (Rule S)
For an abstraction: \[ \hat{w}(\lambda v.M) = \hat{w}(\overline{v}M) \] (Rule abs)

This definition is justified by the following theorem.

(0.37) Theorem. For any variable $v$ and $\lambda$-term $M$,

\[ \lambda v.M = \hat{v}M. \]

Proof. Exercise (by recursion on the structure of $M$). Hint: use extensionality or $\eta$-conversion to introduce dummies as needed, e.g., to rewrite the statement as $(\hat{v}M)N = (\lambda v.M)N$ or as $\lambda u.(\hat{v}M)u = \lambda u.(\lambda v.M)u$.

The following shortcuts can be justified similarly: for any term $M$ with $w \not\in \varphi M$,

\[ \hat{w}M = KM \] (Rule K),
\[ \hat{w}(Mw) = M \] (Rule $\eta$).

The first of these shortcuts subsumes rules K’ and K”.$^2$ Of course, in many cases further ad hoc shortcuts (with similar justification) will present themselves.

For the sake of uniformity, we adopt for an abstraction eliminator $\hat{w}$ the same parentheses convention$^2$ as for an abstractor $\lambda w$.

(0.38) Example. Transforming $T := \lambda xyz.xzy$ into a combinator term. By Theorem 0.37, it suffices to elaborate $\hat{x}\hat{y}\hat{z}xzy$.

\[
\begin{align*}
\hat{x}zy &= \langle \text{Rule S} \rangle S(\hat{x}x)(\hat{y}y) \\
&= \langle \text{Rule } \eta \rangle Sx(\hat{y}y) \\
&= \langle \text{Rule K} \rangle Sx(Ky) \\
\hat{y}Sx(Ky) &= \langle \text{Rule S} \rangle S(\hat{y}Sx)(\hat{y}Ky) \\
&= \langle \text{Rule } \eta \rangle S(\hat{y}Sx)K \\
&= \langle \text{Rule K} \rangle S(K(Sx))K \\
\hat{x}S(K(Sx))K &= \langle \text{Rule S} \rangle S(\hat{x}S(K(Sx)))(\hat{x}K) \\
&= \langle \text{Rule S} \rangle S(S(\hat{x}S)(\hat{x}K(Sx)))(\hat{x}K) \\
&= \langle \text{Rule K} \rangle S(S(KS)(\hat{x}K(Sx)))(KK) \\
&= \langle \text{Rule S} \rangle S(S(KS)(S(\hat{x}K)(\hat{x}Sx)))(KK) \\
&= \langle \text{Rule } \eta \rangle S(S(KS)(S(\hat{x}K)S))(KK) \\
&= \langle \text{Rule K} \rangle S(S(KS)(S(KK)S))(KK)
\end{align*}
\]

$^2$Warning: this convention may differ from the one used in earlier versions of these notes.
Chapter 0. Expressions and Substitution

This example covers all representative cases and makes additional calculation examples redundant. Yet, exercises remain necessary to familiarize the reader with the calculations.

Note that it is often useful to introduce abbreviations for repetitive subterms. Consider, for instance, the lambda term \( Y := \lambda x. (\lambda y. x(y)) (\lambda y. x(y)) \). Transformation of the subterm \((\lambda y. x(y))\) yields \( S(Kx)(SII)\), and calculating \( \hat{x}(S(Kx)(SII)) \) yields \( S(S(KS)K)(K(SII)) \). Since the latter appears twice in the end result, introduce \( X := S(S(KS)K)(K(SII)) \) to write the result as \( SXX \), and finally define \( Y := SXX \). Elaboration of the details is left as an exercise.

0.3.1 De Bruijn numbers

De Bruijn numbers are numbered markers that are used instead of variables, e.g., \((\#0^{a^b}) a b = a^b$$ and \((\#1^{a^b}) a b = b^a$$. An application example is the parameter mechanism of the \( \TeX \) typesetting system, used in preparing these notes.

For readers familiar with the lambda calculus and combinator terms, constructing a syntax and calculation rules for terms based on De Bruijn numbers is more of an exercise in data structures and semantic functions. The interesting parts are the renumbering schemes in defining substitution.

0.3.2 Abstraction in Funmath

The general form of an abstraction in Funmath is \textit{binding.expression} where \textit{binding} has the form \( v:S \) \textbf{with} \( p \) or, more succinctly, \( v:S \land p \). Here \( v \) is an identifier (or identifier tuple), \( S \) a set expression and \( p \) a proposition. The part \( \land p \) is optional. In the abstraction \( v:S \land p.e \), we can see : as replacing the letter \( \lambda \) indicating that one or more identifiers are being introduced (or \textit{bound}). The bound identifiers in an abstraction are called \textit{variables}, and their scope is local.

This notation is chosen because its combination with elastic operators (\( \sum, \forall \), etc.) yields common mathematical notations like \( \forall x : X.p \) and \( \sum i:0..n-1.x^i \).

Note that \( v \) may be a tuple of variables. In \( v:S \land p.e \), the scope of these variables is \( p \) and \( e \), not \( S \). Hence free occurrences of variables from \( v \) in \( S \) are free in \( v:S \land p.e \). Of course, it is very confusing (and hence poor style) to choose the variables in \( v \) in such a way that some of them occur free in \( S \).

By definition, an abstraction of the form \( v:S \land p.e \) denotes a function characterized by the following two axioms:

\begin{itemize}
  \item \textbf{Domain axiom} \( d \in \mathcal{D} (v:S \land p.e) \equiv d \in S \land p^v_d \)
  \item \textbf{Mapping axiom} \( d \in \mathcal{D} (v:S \land p.e) \Rightarrow (v:S \land p.e) d = e^v_d \) (\( \beta \)-conversion)
\end{itemize}

An example is the abstraction \( n:Z \land n \geq 0.2 \cdot n \), which doubles every element in its domain (the set of nonnegative integers, i.e., the set \( \mathbb{N} \) of natural numbers).

In our formalism, a function is fully characterized by its domain and its mapping (nothing else). Hence such an abstraction unambiguously denotes a function. Observe, however, that we have chosen to formulate the axioms using propositional operators such as \( \equiv, \land \) and \( \Rightarrow \). Therefore, further discussion is postponed until the propositional calculus has been formally presented in Chapter 1.
Chapter 1

Calculational Proposition Logic

We introduce proposition calculus as a tool for everyday mathematical practice. Starting with the common concepts of deduction based on axioms and inference rules, we show how the classical proof style can be cast into a calculational format that linearizes the usual tree structure of a proof by recognizing and emphasizing a ‘main line of reasoning’. However, this is only an intermediate stepping stone, and is soon discarded in favor of a truly calculational style, where the logical operators themselves appear as the links in the chain of calculation.

These propositional operators are introduced axiomatically and one at a time, starting with the basic axioms for implication ($\Rightarrow$) —which allows introducing the deduction theorem at the earliest possible stage— followed by negation ($\neg$). Although these two operators in principle constitute a complete set, the introduction of other, derived operators ($\equiv, \land, \lor$) makes practical use more convenient. For the same reason, we also provide appropriate nomenclature for the most often-used calculation laws that are derived from the axioms.

Finally, we outline binary algebra as a concrete model of proposition calculus, and conclude by deriving convenient calculation rules for conditional expressions.

1.0 Propositions, basic operators and calculation rules

1.0.0 The (sub)language of propositions

The calculus of propositions does not depend on the form of the expressions with a higher level of detail than the propositions themselves. Hence we take the (sub)language of propositions to consist of expressions whose syntax is defined according to Definition 0.2 on page 2, containing only propositional variables, propositional constants, and applications of propositional operators.

\[
\begin{align*}
\text{proposition} & ::= \text{variable} \mid \text{constant} \mid \text{implication} \\
\text{application} & ::= (\text{cop}_1 \text{ proposition}) \mid (\text{proposition cop}_2 \text{ proposition})
\end{align*}
\]

The (propositional) variables are chosen near the end of the alphabet, e.g., $x$, $y$, $z$. The constants and operators will be introduced one at a time as we proceed.

As before, lowercase letters like $p$, $q$, $r$ or in that vicinity are metavariables standing for propositional expressions.
1.0.1 Implication and deduction rules

We start with implication (⇒) as the only operator. The advantage is having the deduction theorem available early, but equality (Leibniz’s principle) is delayed.

An expression of the form \((p \Rightarrow q)\) is an implication with antecedent \(p\) and consequent \(q\). Certain parentheses are made optional by the following rules.

- **Omitting outer parentheses**: only for implications standing by themselves.
- **Right association**: \(p \Rightarrow q \Rightarrow r\) stands for \(p \Rightarrow (q \Rightarrow r)\). Warning: the same care has to be taken here as with \(LMN = (LM)N\) in lambda calculus.

A. Modus ponens, axioms and their soundness

The proposition calculus has only two inference rules:

\[(1.0) \text{ INFERENCE RULES: INSTANTIATION AND MODUS PONENTS} \]

- **(INS) Instantiation**: \(\frac{p}{p[v := q]}\) (provided \(p\) is a theorem)
- **(MP) Modus Ponens**: \(\frac{p \Rightarrow q}{q}\) (any propositions \(p\) and \(q\))

The calculation rules for “⇒” are fully specified by two axioms.

\[(1.1) \text{ AXIOMS FOR “⇒”: WEAKENING AND (LEFT) DISTRIBUTIVITY} \]

- **(W⇒) Weakening**: \(x \Rightarrow y \Rightarrow x\)
- **(D⇒) (Left) Distributivity**: \((x \Rightarrow y \Rightarrow z) \Rightarrow (x \Rightarrow y) \Rightarrow (x \Rightarrow z)\)

Inference rules and axioms are used for deduction in the following way.

\[(1.2) \text{ DEFINITION: THEOREMS} \quad q \text{ is a theorem, written } \vdash q, \text{ if } q \text{ is either} \]

- (i) an axiom, or
- (ii) the conclusion of an inference rule with theorems for premises.

A (formal) proof of \(q\) is a systematic record of how \(\vdash q\) is established.

\[(1.3) \text{ DEFINITION: DEDUCTION} \quad \text{If } \mathcal{H} \text{ is a collection of propositions (called hypotheses here), then } q \text{ is a consequence of } \mathcal{H}, \text{ written } \mathcal{H} \vdash q, \text{ if } q \text{ is either} \]

- (i) a hypothesis in \(\mathcal{H}\), or
- (ii) the conclusion of INS where the premises are theorems, or
- (iii) the conclusion of MP where the premises are consequences of \(\mathcal{H}\).

A (formal) deduction of \(q\) from \(\mathcal{H}\) is a record of how \(\mathcal{H} \vdash q\) is established.

Certain technical terms sound rather similar, so we recall them here together:

- \(p \Rightarrow q\) is an implication with antecedent \(p\) and consequent \(q\).
- \(\mathcal{H} \vdash q\) is a sequent with hypotheses \(\mathcal{H}\) and consequent \(q\).
- \(\frac{\mathcal{P}}{q}\) is an inference rule with premises \(\mathcal{P}\) and conclusion \(q\). A synonym for conclusion is direct consequence.
1.0. Propositions, basic operators and calculation rules

If $\mathcal{H} \vdash q$ for empty $\mathcal{H}$, then $q$ is a theorem, and we write $\mathcal{H} \vdash q$ as $\vdash q$. Being a theorem or a consequence of the hypotheses is called the \textit{status} of a proposition.

Note that no reference whatsoever is made to meaning. Of course, in the standard interpretation, formulas denote truth values. For reasons given later, we choose $\mathbb{B} := \{0,1\}$ for the set of truth values.

The soundness of the deduction scheme in Definition 1.3 w.r.t. the standard interpretation is verified as follows. We take $\mathbb{B}$ to be the domain of interpretation, and assign $\Rightarrow$ the usual interpretation, viz. $k_2 \Rightarrow (x,y) = (1,y)$ or, equivalently, $k_2 \Rightarrow (0,y) = 1$ and $k_2 \Rightarrow (1,y) = y$ for any $(x,y) : \mathbb{B}^2$. The axioms were chosen to be \textit{tautologies}, i.e., they evaluate to 1 in any state (verification left as an exercise\(^6\)). The modus ponens has the property that $\mathcal{E} q$ s evaluates to 1 in every state in which both $\mathcal{E} [p \Rightarrow q]$ s and $\mathcal{E} p$ s evaluate to 1. Therefore, $\mathcal{H} \vdash p$ guarantees that $\mathcal{H} \vdash p$. Note that MP is not a strict inference rule: the premises need not be theorems. Hence the expressions in $\mathcal{H}$ need not be theorems either.

B. Traditional styles for organizing formal proofs

In traditional texts on formal logic, a proof or deduction is a numbered list of formulas, each of which is a hypothesis (if any), or an instantiation of a theorem, or the conclusion an inference rule with premises earlier in the list. Which of these justifications is applicable is stated next to (preferably before) the formula, referring to the premises in some suitable way (name, pointer in the list etc.).

\begin{enumerate}
\item Example: a proof for the theorem $x \Rightarrow x$
\begin{enumerate}
\item INS $D \Rightarrow (x \Rightarrow (x \Rightarrow x)) \Rightarrow (x \Rightarrow x) \Rightarrow (x \Rightarrow x)$
\item INS $W \Rightarrow x \Rightarrow (x \Rightarrow x) \Rightarrow x$
\item MP $0,1 \Rightarrow (x \Rightarrow x) \Rightarrow (x \Rightarrow x)$
\item INS $W \Rightarrow x \Rightarrow x \Rightarrow x$
\item MP $2,3 \Rightarrow x \Rightarrow x$
\end{enumerate}
\end{enumerate}

Other texts use \textit{sequents}, directly reflecting the tree structure without pointers. One writes $\mathcal{H} \vdash q$ as $\frac{\mathcal{H}}{q}$ to save horizontal space, and writes the inference rule next to it, say, I for instantiation and M for modus ponens.

\begin{enumerate}
\item Example: a proof for the theorem $x \Rightarrow x$ in sequent style
\begin{align*}
& (x \Rightarrow y \Rightarrow z) \Rightarrow (x \Rightarrow y) \Rightarrow x \Rightarrow z \\
& (x \Rightarrow (x \Rightarrow x) \Rightarrow x) \Rightarrow (x \Rightarrow x) \Rightarrow x \Rightarrow x \\
& x \Rightarrow y \Rightarrow x \\
& x \Rightarrow (x \Rightarrow x) \Rightarrow x \\
& (x \Rightarrow x) \Rightarrow x \\
& x \Rightarrow x
\end{align*}
\end{enumerate}

C. A calculational format for the traditional proof style

The traditional organization of formal proofs as numbered lists of formulas, and the various forms of \textit{sequent calculus}, are rather clumsy, because they introduce excessive nomenclature and obscure the structure of reasoning. This is why they

\(6\) Another exercise consists in showing that the interpretation given for $\Rightarrow$ corresponds to the only nontrivial operator from $\mathbb{B}^2$ to $\mathbb{B}$ that makes the weakening axiom a tautology.
are used only in the introductory part of logic textbooks, and not beyond. Although our aim is far from encouraging these styles in any form, we show a vastly improved form by casting them into a calculational format. This is only for “bootstrapping” towards a true calculational style, and is **discarded afterwards**.

Observe that the recursive definition of consequence makes a deduction into a tree structure where the consequence is the root and the invoked axioms and hypotheses are leaves. Forking branches correspond to using Modus Ponens. Subtrees can be separated as lemmata (that can be granted as leaves).

A **calculational deduction** is in essence a deduction in which a path from a leaf to the root is taken as the principal chain of reasoning that conveys the crucial steps, and subtrees or leaves are just auxiliary deductions. The desired format, avoiding repetition of expressions and highlighting the line of reasoning, is

\[
p_0 \quad R_0 \quad \langle \text{Justification}_0 \rangle \quad p_1
\]
\[
R_1 \quad \langle \text{Justification}_1 \rangle \
\]

where \( R_k \) is the relation between \( p_k \) and \( p_{k+1} \). In the true calculational style as established soon, it is expressed by an operator in the language \( (\Rightarrow) \). However, the rules to support this (e.g., transitivity of \( \Rightarrow \)) must be derived first.

Meanwhile, we cast the traditional style in calculational format, using just formalized comments or **labels** instead of relational operators. Chaining is done by modus ponens, similar to transitivity. Since the two premises \( p \) and \( p \Rightarrow q \) have different form, we emphasize this by using different labels, \( \downarrow \) and \( \times \).

\[
\langle \text{Deduction steps for} \ p \rangle \
\downarrow \langle \text{Justification for} \ p \Rightarrow q \rangle \
\times \langle \text{Justification for} \ p \rangle
\]

Label \( \downarrow \) announces that the justification establishes the implication \( p \Rightarrow q \) between two successive lines, and \( \times \) (“crossing out”) that the justification establishes the antecedent \( p \) of the proposition \( p \Rightarrow q \) in the preceding line. A justification is either a hypothesis\(^1\) or a theorem that instantiates to \( p \Rightarrow q \) or \( p \). Although only the theorems themselves are stated, their instantiations can be read in the chain. The first expression of a chain is justified similarly.

We illustrate this in the proof below. Compare the amount of writing with the traditional forms in example (1.4) and (1.5), although no information is lost.

\textbf{(1.6) Theorem: (R}\Rightarrow\text{) Reflexivity of implication} \quad \boxed{x \Rightarrow x}

\textbf{Proof}

\[
\langle \text{W} \Rightarrow \rangle \quad x \Rightarrow (x \Rightarrow x) \Rightarrow x
\]
\[
\downarrow \langle \text{D} \Rightarrow \rangle \quad (x \Rightarrow x \Rightarrow x) \Rightarrow x \Rightarrow x
\]
\[
\times \langle \text{W} \Rightarrow \rangle \quad x \Rightarrow x
\]

\textbf{D. Further Theorems Leading to a Fully Calculational Style}

The following theorems are intended to introduce calculational chaining in the algebraic style, i.e., where the links are the operators of the calculus (thus far, just \( \Rightarrow \)) instead of the meta-operator \( \vdash \).

\(^1\)Keep in mind that hypotheses that are not theorems should not be instantiated.
As we have already observed for equality, the basis for chaining is transitivity.

**1.7 Metatheorem: (T⇒) Transitivity of Implication**

\[ p \Rightarrow q, \ q \Rightarrow r \quad \vdash \quad p \Rightarrow r \]

**Proof**

\[ \langle W⇒ \rangle \quad (q \Rightarrow r) \Rightarrow p \Rightarrow q \Rightarrow r \]

\[ \times \quad \langle q \Rightarrow r \rangle \quad p \Rightarrow q \Rightarrow r \]

\[ \downarrow \quad \langle D⇒ \rangle \quad (p \Rightarrow q) \Rightarrow (p \Rightarrow r) \]

\[ \times \quad \langle p \Rightarrow q \rangle \quad p \Rightarrow r \]

This justifies chaining deduction steps by writing \( p \Rightarrow q, \ q \Rightarrow r \vdash p \Rightarrow r \) as

\[ p \Rightarrow \langle \text{Justification for } p \Rightarrow q \rangle \quad q \]

\[ \Rightarrow \langle \text{Justification for } q \Rightarrow r \rangle \quad r \]

Note that a calculational proof for \( p \Rightarrow q \) can serve two purposes: just proving \( p \Rightarrow q \), or providing a deduction for \( p \vdash q \) by observing that \( p \vdash (p \Rightarrow q) \ q \). Hence a link of the form \( p \Rightarrow (p \Rightarrow q) \ q \) subsumes \( p \vdash (p \Rightarrow q) \ q \).

A first application example of \( T⇒ \) consists in letting transitivity bootstrap itself from a metatheorem to a formula.

**1.8 Theorem: (M⇒) (Right) Monotonicity of Implication**

\[
\begin{array}{c}
(x \Rightarrow y) \Rightarrow (z \Rightarrow x) \Rightarrow (z \Rightarrow y)
\end{array}
\]

**Proof**

\[ x \Rightarrow y \Rightarrow \langle W⇒ \rangle \quad z \Rightarrow x \Rightarrow y \]

\[ \Rightarrow \langle D⇒ \rangle \quad (z \Rightarrow x) \Rightarrow (z \Rightarrow y). \]

**Corollary: Metatheorem (WC), Weakening the Consequent**

\[ p \Rightarrow q \quad \vdash \quad (r \Rightarrow p) \Rightarrow (r \Rightarrow q) \]

A first application of \( M⇒ \) appears in the proof of the following theorem, which will enable us to subsume all \( \times \)-steps.

**1.9 Theorem: (P⇒) Modus Ponens as a Formula with “⇒”**

\[
\begin{array}{c}
x \Rightarrow (x \Rightarrow y) \Rightarrow y
\end{array}
\]

**Proof**

We start with an instantiation of the theorem \( R⇒ \).

\[ \langle R⇒ \rangle \quad (x \Rightarrow y) \Rightarrow x \Rightarrow y \]

\[ \Rightarrow \quad \langle D⇒ \rangle \quad ((x \Rightarrow y) \Rightarrow x) \Rightarrow ((x \Rightarrow y) \Rightarrow y) \]

\[ \Rightarrow \quad \langle M⇒ \rangle \quad (x \Rightarrow (x \Rightarrow y) \Rightarrow x) \Rightarrow (x \Rightarrow (x \Rightarrow y) \Rightarrow y) \]

Since the first formula in this deduction is a theorem, so is the last formula, say \( q \), and \( q \times \langle W⇒ \rangle \ x \Rightarrow (x \Rightarrow y) \Rightarrow y \) completes the proof.
Chapter 1. Calculational Proposition Logic

Corollary A link of the form \( p \Rightarrow q \Rightarrow \langle p \rangle\) subsumes \( p \Rightarrow q \times \langle p \rangle\).  
Proof: \( p \Rightarrow (p \Rightarrow q) \Rightarrow q \Downarrow \langle p \rangle \quad (p \Rightarrow q) \Rightarrow q.\)  

Now both \( \times \) and \( \Downarrow \) have been subsumed by implications. Note that MP itself remains necessary in order to validate the earlier observation that a deduction for \( p \Rightarrow q \) can also serve as a deduction from \( p \) to \( q \). The converse is certainly less evident, but also holds, as shown later by the deduction theorem.

The interchangeability of the premisses in the transitivity metatheorem \((T \Rightarrow)\) appears lost in \((M \Rightarrow)\), but can be recovered by means of the following theorem which, by its generality, will be used quite often in the future.

(1.10) Theorem: \((S \Rightarrow)\) Shunting with implication

\[
(x \Rightarrow y \Rightarrow z) \Rightarrow y \Rightarrow x \Rightarrow z
\]

Proof

\[
x \Rightarrow y \Rightarrow z \Rightarrow (D \Rightarrow) (x \Rightarrow y) \Rightarrow x \Rightarrow z
\]
\[
\Rightarrow (M \Rightarrow) (y \Rightarrow x \Rightarrow y) \Rightarrow y \Rightarrow x \Rightarrow z
\]
\[
\Rightarrow (W \Rightarrow) y \Rightarrow x \Rightarrow z
\]

Corollary: \((A \Rightarrow)\) (Left) Antimonotonicity of \( \Rightarrow \)

\[
(x \Rightarrow y) \Rightarrow (y \Rightarrow z) \Rightarrow (x \Rightarrow z)
\]

Proof Right monotonicity yields left antimonotonicity by shunting.

Corollary: \((SA)\) Strengthening the antecedent

\[
p \Rightarrow q \vdash (q \Rightarrow r) \Rightarrow (p \Rightarrow r)
\]

Proof \( (p \Rightarrow q) \Rightarrow (A \Rightarrow) (q \Rightarrow r) \Rightarrow (p \Rightarrow r). \)

The proofs of the preceding theorems seem to require some puzzling in the beginning, but go smoother after some practice when certain patterns become apparent. In fact, most of these patterns are subsumed by the deduction theorem.

E. The Deduction Theorem

In informal reasoning, \( p \Rightarrow q \) is often proved by assuming \( p \) (as hypothesis) and deducing \( q \), i.e., providing a deduction for \( p \vdash q \). Obviously, this is not the same. Moreover, a deduction for \( p \Rightarrow q \) is usually much longer.

Yet, the deduction theorem allows us to use the shorter deduction by showing that the existence of a deduction for \( p \vdash q \) implies the existence of a deduction for \( p \Rightarrow q \). Even more: the proof of the deduction theorem is constructive in the sense that it provides an algorithm for transforming a deduction for \( p \vdash q \) into a deduction for \( p \Rightarrow q \).
Hence $p \vdash q$ makes $p \Rightarrow q$ a theorem, although neither $p$ nor $q$ need be theorems. The method of proving $p \Rightarrow q$ by assuming $p$ and deducing $q$ is called assuming the antecedent.

To provide a more general setting, we write $\mathcal{H} \land p$ for the collection of hypotheses augmented by $p$, not specifying whether $\mathcal{H}$ is a list or a set.

We first show the converse of the theorem as a a simpler warming-up exercise.

(1.11) Theorem: Converse of the Deduction Theorem
If $\mathcal{H} \vdash p \Rightarrow q$ then $\mathcal{H} \land p \vdash q$.

Proof: A deduction for $\mathcal{H} \vdash p \Rightarrow q$ is a deduction for $\mathcal{H} \land p \vdash p \Rightarrow q$. Adding the single step $\times \langle p \rangle \Rightarrow (p \Rightarrow q)$ yields a deduction for $\mathcal{H} \land p \vdash q$.

(1.12) Theorem: Deduction Theorem
If $\mathcal{H} \land p \vdash q$ then $\mathcal{H} \vdash p \Rightarrow q$.

Proof: Let $D$ be a deduction for $\mathcal{H} \land p \vdash q$. We transform $D$ recursively into a deduction for $\mathcal{H} \vdash p \Rightarrow q$. We consider the last step of $D$, for which there are three possibilities. In the first two cases, $D$ consists of a single step only, and is used only in the trivial sense in the transformation.

(a) If $q$ is in $\mathcal{H} \land p$, then we consider two subcases.
   a. If $q$ is in $\mathcal{H}$, then a deduction for $\mathcal{H} \vdash p \Rightarrow q$ is
      \[
      \langle W \Rightarrow \rangle \quad q \Rightarrow p \Rightarrow q
      \]
      \[
      \times \quad \langle \text{Hyp } q \rangle \quad p \Rightarrow q
      \]
   b. If $q$ is $p$, then the deduction for $\text{R} \Rightarrow$ is a deduction for $\mathcal{H} \vdash p \Rightarrow q$.
   (ii) If $q$ is an axiom, then we proceed as in (i).a.
   (iii) If, in $D$, $q$ is a direct consequence of $r \Rightarrow q$ and $r$ by MP, then $D$ contains subdeductions for $\mathcal{H} \land p \vdash r \Rightarrow q$ and $\mathcal{H} \land p \vdash r$. These we first transform into deductions for $\mathcal{H} \vdash r \Rightarrow q$ and $\mathcal{H} \vdash r \Rightarrow r$. The results are combined in the following deduction for $\mathcal{H} \vdash p \Rightarrow q$.

\[
\langle D \Rightarrow \rangle \quad (p \Rightarrow r \Rightarrow q) \Rightarrow (p \Rightarrow r) \Rightarrow (p \Rightarrow q)
\]
\[
\times \quad \langle p \Rightarrow r \Rightarrow q \rangle \quad (p \Rightarrow r) \Rightarrow (p \Rightarrow q)
\]
\[
\times \quad \langle p \Rightarrow r \rangle \quad (p \Rightarrow q)
\]

In the proof, we used $\times$ and $\downarrow$ to show that the deduction theorem is independent of the preceding theorems (except $\text{R} \Rightarrow$). This allows illustrating its application by providing even shorter proofs for these theorems.

Note that, by recursion, proving $p, q, r \vdash s$ suffices to prove $p \Rightarrow q \Rightarrow r \Rightarrow s$.

(1.13) Examples: using the deduction theorem to provide shorter proofs
We consider successively $(T \Rightarrow), (M \Rightarrow), (P \Rightarrow)$ and $(S \Rightarrow)$.

- To establish $p \Rightarrow q, q \Rightarrow r \vdash p \Rightarrow r$, we prove $p \Rightarrow q, q \Rightarrow r, p \vdash r$:

\[
\langle p \rangle \quad p \downarrow \langle p \Rightarrow q \rangle \quad q \downarrow \langle q \Rightarrow r \rangle \quad r
\]
• For \((x \Rightarrow y) \Rightarrow (z \Rightarrow y) \Rightarrow (x \Rightarrow y)\), prove \(x \Rightarrow y, z \Rightarrow y, x \vdash y\) as before or, calculationally (since we now have \(T \Rightarrow\) for chaining),

\[
x \Rightarrow \langle x \Rightarrow y \rangle \quad y \Rightarrow \langle y \Rightarrow z \rangle \quad z
\]

Note that the distinction between the deductions for \((T \Rightarrow)\), \((M \Rightarrow)\) and \((A \Rightarrow)\) is blurred, but the statements remain different. Observe also that the theorems can be presented in a different order.

• To establish \(x \Rightarrow (x \Rightarrow y) \Rightarrow y\), prove \(x \Rightarrow y, x \vdash y\) (trivial).

• To establish \((x \Rightarrow y \Rightarrow z) \Rightarrow y \Rightarrow x \Rightarrow z\), prove \((x \Rightarrow y \Rightarrow z), x, y \vdash z\) (exercise).

**Important remark** From the preceding examples, one should not obtain the impression that the deduction theorem always yields such spectacular simplifications. Indeed, the following must be kept in mind.

• The theorems thus far were mainly “rearrangements” in chained implications, so they lent themselves well to application of the deduction theorem.

• The deduction theorem “works” only for implications, not other operators.

• Using the deduction theorem does not always yield the most elegant proof.

### 1.0.2 Introducing the the truth constant

First we observe that, strictly speaking, we do not need a truth constant, since any theorem can fulfil this role.

\[1.14 \text{ LEMMA: THEOREMS AS RIGHT ZERO AND LEFT IDENTIY “⇒”}
\]

Any theorem \(t\) has the following properties.

\[a. \text{ Right zero: } x \Rightarrow t \text{ (also: } t \Rightarrow (x \Rightarrow t) \text{ and } (x \Rightarrow t) \Rightarrow t)\]

\[b. \text{ Left identity: } x \Rightarrow (t \Rightarrow x) \text{ and } (t \Rightarrow x) \Rightarrow x\]

**Proof** a. follows from \((W⇒)\) and MP, the first part of b. from \((W⇒)\), the second part from instantiating \((P⇒)\) as \(t \Rightarrow (t ⇒ x) ⇒ x\) and MP.

In view of the algebraic use of ⇒, the embedding in the remainder of mathematics (in particular, arithmetic) and the introduction of equality, we introduce the constant \(\bot\) and specify its properties by the simplest axiom imaginable.

\[1.15 \text{ AXIOM: THE TRUTH CONSTANT “} 1 “
\]

The preceding lemma directly yields the following properties.

\[1.16 \text{ THEOREM: 1 AS RIGHT ZERO AND LEFT IDENTITY “⇒”}
\]

\[a. \text{ Right zero: } x ⇒ 1 \text{ (also: } 1 ⇒ (x ⇒ 1) \text{ and } (x ⇒ 1) ⇒ 1)\]

\[b. \text{ Left identity: } x ⇒ (1 ⇒ x) \text{ and } (1 ⇒ x) ⇒ x\]
1.0.3 Negation: axiom, calculation rules, falsehood constant

The negation operator \( \neg \) is a 1-place function symbol. To avoid accumulation of parentheses as in \( \neg(\neg(\neg p)) \), we write \( \neg^n p \) for \( n \)-fold application of \( \neg \) to \( p \); formally: \( (\neg^0 p) \) stands for \( p \) and \( (\neg^{n+1} p) \) for \( \neg(\neg^n p) \). The only basic rule for negation (in addition, of course, to the rules for implication) is the following.

\[
\text{(1.17) Axiom: (CP\Rightarrow) Contrapositive \quad (\neg x \Rightarrow \neg y) \Rightarrow y \Rightarrow x}
\]

Combined with the first axiom for implication, this yields the following theorem.

\[
\text{(1.18) Theorem: (CA\Rightarrow) Contradictory antecedents \quad \neg x \Rightarrow \neg x \Rightarrow y}
\]

**Proof** \( \neg x \Rightarrow (W\Rightarrow) \neg y \Rightarrow \neg x \Rightarrow (\text{CP}\Rightarrow) \ x \Rightarrow y. \)

By shunting, this also yields \( x \Rightarrow \neg x \Rightarrow y. \) A first consequence of (CA) is that, from contradictory hypotheses, anything can be deduced (“ex falso sequitur quodlibet”). This principle is captured by the following corollary.

**Corollary: Contradictory hypotheses \quad p, \neg p \vdash q**

Observe that \( x \Rightarrow \neg x \Rightarrow x \) is trivial as an instantiation of weakening. The converse, namely \( (\neg x \Rightarrow x) \Rightarrow x, \) is somewhat tricky to prove but, once established, it makes the proof of every other theorem about negation rather straightforward.

\[
\text{(1.19) Theorem: (SI\Rightarrow) Skew idempotency of “\Rightarrow” \quad (\neg x \Rightarrow x) \Rightarrow x}
\]

**Proof** We start by instantiating (CA\Rightarrow) as \( \neg x \Rightarrow x \Rightarrow \neg(\neg x \Rightarrow x). \)

\[
\neg x \Rightarrow x \Rightarrow (D\Rightarrow) \quad (\neg x \Rightarrow x) \Rightarrow \neg x \Rightarrow \neg(\neg x \Rightarrow x)
\]

\[
\Rightarrow (\text{WC by CP\Rightarrow}) \quad (\neg x \Rightarrow x) \Rightarrow (\neg x \Rightarrow x) \Rightarrow x
\]

\[
\Rightarrow (\text{AB\Rightarrow}) \quad (\neg x \Rightarrow x) \Rightarrow x
\]

Proving (AB\Rightarrow) Absorption, i.e., \( (x \Rightarrow x \Rightarrow y) \Rightarrow x \Rightarrow y, \) is left as an exercise.

A first application of (SI\Rightarrow) occurs in the proof of the following theorem, which fulfills some reasonable expectations regarding negation.

\[
\text{(1.20) Theorem: (DN\Rightarrow) Double negation \quad a. \quad -^2 x \Rightarrow x \quad b. \quad x \Rightarrow -^2 x}
\]

**Proof**  

a. Proof for \( -^2 x \Rightarrow x. \)

\[
-^2 x \Rightarrow (W\Rightarrow) \quad -x \Rightarrow -^2 x
\]

\[
\Rightarrow (\text{CP\Rightarrow}) \quad -x \Rightarrow x
\]

\[
\Rightarrow (\text{SI\Rightarrow}) \quad x
\]

b. By instantiating a.: \( (\text{a.}) \quad -^3 x \Rightarrow \neg x \Rightarrow (\text{CP\Rightarrow}) \quad x \Rightarrow -^2 x. \)

In referencing, we designate \( -^2 x \Rightarrow x \) as “double negation antecedent” (DNA), and \( x \Rightarrow -^2 x \) as “double negation consequent” (DNC).

This theorem allows to prove a simple variant of (CP\Rightarrow).
(1.21) **Theorem:** (CPR ⇒) **Reversed contrapositive**

\[(x \Rightarrow y) \Rightarrow (\neg y \Rightarrow \neg x)\]

**Proof**

\[x \Rightarrow y \Rightarrow (\text{SA by DNA}) \quad \neg^2 x \Rightarrow y\]
\[\Rightarrow (\text{WC by DNC}) \quad \neg^2 x \Rightarrow \neg^2 y\]
\[\Rightarrow (\text{CP ⇒}) \quad (\neg y \Rightarrow \neg x)\]

A second application of (SI ⇒) is bootstrapping (CP ⇒) into a slightly stronger form that will have a very powerful property as a consequence.

(1.22) **Theorem:** (SCP) **Strengthened contrapositive**

\[(\neg x \Rightarrow \neg y) \Rightarrow (\neg x \Rightarrow y) \Rightarrow x\]

**Proof**

\[\neg x \Rightarrow \neg y \Rightarrow (\text{CP ⇒}) \quad y \Rightarrow x\]
\[\Rightarrow (\text{M ⇒}) \quad (\neg x \Rightarrow y) \Rightarrow \neg x \Rightarrow x\]
\[\Rightarrow (\text{WC by SI ⇒}) \quad (\neg x \Rightarrow y) \Rightarrow x\]

(1.23) **Theorem:** (DL ⇒) **Dilemma**

\[\neg x \Rightarrow y \Rightarrow (x \Rightarrow y) \Rightarrow y\]

The proof is left as an exercise.

To conclude, we introduce the **falsehood constant** \(\neg 0\), with the following axiom.

(1.24) **Axiom:** The falsehood constant \(\neg 0\)

(1.25) **Theorem:** Some simple properties of \(0\)

a. \(0 \Rightarrow x\)

b. \((x \Rightarrow 0) \Rightarrow \neg x \text{ and } \neg x \Rightarrow (x \Rightarrow 0)\)

c. \(1 \Rightarrow \neg 0 \text{ and } \neg 1 \Rightarrow 0\)

Of course, there exist many other laws for negation and implication, but we have presented the core material that makes the derivation of such laws very easy when the need arises. Further shortcuts are provided by the techniques explained next.

### 1.0.4 Case analysis and a variant of Shannon expansion

The reader will have correctly guessed that the list of theorems is a never-ending story, even if we restrict ourselves to the properties that are bound to be used frequently in practice. Indeed, experience has shown that, in the course of a formal proof, one often needs a property that is not included in the list anyway.

However, we present a general-purpose technique, called **case analysis**, for handling certain kinds of problems very rapidly and, after a little practice, often by inspection. Typical applications are: checking the correctness of a conjecture,
finding a counterexample if the conjecture is incorrect, and even “patching” incorrect conjectures. A more elegant derivation can then be constructed afterwards.

Since this technique is based on some earlier theorems whose proofs were left as exercises, it does not come entirely free. In tests and exams, it should therefore be used only if the student either has explicit permission to do so, or first provides the missing proofs for all necessary theorems!

(1.26) **Lemma: Binary Cases** For any proposition \( p \) and variable \( v \),

- **(0-case)** \( \vdash \neg v \Rightarrow p \mid_0 \Rightarrow p \) and \( \vdash \neg v \Rightarrow p \Rightarrow p \mid_0 \)
- **(1-case)** \( \vdash v \Rightarrow p \mid_1 \Rightarrow p \) and \( \vdash v \Rightarrow p \Rightarrow p \mid_1 \)

The proof requires induction, and will be left as a later exercise. Variants can be obtained by shunting. From this lemma, we can derive the following metatheorem, which provides a simple and powerful proof technique.

(1.27) **Theorem: Case Analysis** For any proposition \( p \) and variable \( v \),

\( \vdash p \mid_0 \Rightarrow p \mid_1 \Rightarrow p \) and, equivalently, \( p \mid_0 \land p \mid_1 \vdash p \)

In other words, to prove \( p \), it suffices proving \( p \mid_0 \) and \( p \mid_1 \). After a little practice, this can be done by inspection. This is especially useful for verifying conjectures.

An opportunity for appreciating the power of case analysis occurs in the proof of the following little-known variant of a theorem whose standard variant (presented later) is attributed to Shannon.

(1.28) **Theorem: Shannon Expansion with Implication** For any proposition \( p \) and variable \( v \), the following metatheorems hold.

- **Accumulation:** \( \vdash (\neg v \Rightarrow p \mid_0) \Rightarrow (v \Rightarrow p \mid_1) \Rightarrow p \)
- **Weakening:** \( \vdash \neg v \Rightarrow p \mid_0 \) and \( \vdash v \Rightarrow p \mid_1 \)

**Proof** The accumulation part is most easily proven by case analysis on the expression as a whole, i.e., by proving \( ((\neg v \Rightarrow p \mid_0) \Rightarrow (v \Rightarrow p \mid_1) \Rightarrow p) \mid_0 \) and \( (\neg v \Rightarrow p \mid_0) \Rightarrow (v \Rightarrow p \mid_1) \Rightarrow p) \mid_1 \), observing that \( p \mid_0 \land p \mid_1 \) in case \( v \) does not occur in \( e \) (proof by induction left as a later exercise). The weakening part is obtained from Lemma 1.26 by shunting.

### 1.1 Introducing additional operators

#### 1.1.0 Logical equivalence as propositional equality

**A. The Logical Equivalence Operator “≡”**

The syntax of the calculus is extended with the *logical equivalence* operator “≡” (a 2-argument function symbol). The convention for making parentheses optional is that “≡” has the lowest precedence among the propositional operators. Hence, for instance, \( p \Rightarrow q \equiv r \) stands for \( (p \Rightarrow q) \equiv r \).

The standard semantics with domain of interpretation \( \mathbb{B} \) is equality.
Here we focus on the axioms introducing the formal calculation rules. An important observation is that the calculus with just “⇒” and “¬” is essentially complete (see the exercises); there is no need for adding more structure to the algebra. However, new operators can significantly improve convenience.

A common way for introducing rules for new operators that do not add structure to the algebra is via abbreviation, saying that an application of the new operator stands for an expression containing pre-existing operators only, e.g., that \( x \land y \) is an abbreviation for \( \neg(x \Rightarrow \neg y) \), and \( x \equiv y \) for \( (x \Rightarrow y) \land (y \Rightarrow x) \).

To avoid this style breach, we present the axioms in the same way as the preceding ones, which at the same time better reveals their purpose.

(1.29) **Axioms for Logical Equivalence ("≡")**

- (AS⇒) Antisymmetry of ⇒: \( (x \Rightarrow y) \Rightarrow (y \Rightarrow x) \Rightarrow (x \equiv y) \)
- (W≡) Weakening of ≡: \( (x \equiv y) \Rightarrow x \Rightarrow y \) and \( (x \equiv y) \Rightarrow y \Rightarrow x \)

This simplicity has its price. Indeed, letting the application of the new operator appear as the consequent in one axiom (say, \( q \Rightarrow p \)) and as the antecedent in another (say, \( p \Rightarrow r \)) yields (by transitivity) a theorem involving the old operators only (say, \( q \Rightarrow r \)), which must be consistent with the existing calculus (exercise: give an example of a pair of axioms that causes an inconsistency of this kind).

However, here the axioms have a structure that allows a simple check for consistency and even for a certain kind of uniqueness (although the latter is not a primary concern). To appreciate this, observe that the general form of this system of axioms is \( p \Rightarrow q \Rightarrow x \ast y \) and \( x \ast y \Rightarrow p \) and \( x \ast y \Rightarrow q \). This can be seen as a system of of implications, namely

(1.30) \[ p \Rightarrow q \Rightarrow s \quad s \Rightarrow p \quad s \Rightarrow q. \]

with unknown (expression) \( s \). Analysis is facilitated by the following theorem about the expression \( \neg(x \Rightarrow \neg y) \) containing only the operators “⇒” and “¬”.

(1.31) **Theorem: Accumulation and Weakening of \( \neg(x \Rightarrow \neg y) \)**

- Accumulation: \( x \Rightarrow y \Rightarrow \neg(x \Rightarrow \neg y) \)
- Weakening: \( \neg(x \Rightarrow \neg y) \Rightarrow x \) and \( \neg(x \Rightarrow \neg y) \Rightarrow y \)

These properties will be captured later on by the conjunction operator “\&”. Proofs for this theorem and for most of the following ones are left as exercises.

(1.32) **Theorem: Existence and Uniqueness of Solutions to**

\[ p \Rightarrow q \Rightarrow s \quad s \Rightarrow p \quad s \Rightarrow q \] (with unknown \( s \))

- Existence: \( s := \neg(p \Rightarrow \neg q) \) is a solution.
- Uniqueness: if \( s \) and \( s' \) are solutions, then \( s \equiv s' \).

One can show also (exercise) that the pair of implications \( s \Rightarrow p \) and \( s \Rightarrow q \) is interchangeable with the single implication \( s \Rightarrow \neg(p \Rightarrow \neg q) \).
(1.33) **Theorem**: Every solution \( s \) to the system \( p \Rightarrow q \Rightarrow s, \ s \Rightarrow p \) has the following properties.

\[
(s \Rightarrow r) \Rightarrow (p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow q) \Rightarrow r
\]

**Corollary**: Calculation properties of \( \equiv \)

\[
((x \equiv y) \Rightarrow z) \Rightarrow (x \Rightarrow y) \Rightarrow (y \Rightarrow x) \Rightarrow z
\]

\[
((x \Rightarrow y) \Rightarrow (y \Rightarrow x) \Rightarrow z) \Rightarrow (x \equiv y) \Rightarrow z
\]

**B. Propositional equality and equational theorems**

The logical equivalence operator has all the formal properties of equality. The proofs are left as exercises (for \( \Leftrightarrow \), use structural induction).

(1.34) **Theorem**: **Logical equivalence as propositional equality**

a. The logical equivalence operator \( \equiv \) is an equivalence relation.

* (R\(\equiv\)) Reflexivity: \( x \equiv x \)
* (S\(\equiv\)) Symmetry: \( (x \equiv y) \Rightarrow (y \equiv x) \)
* (T\(\equiv\)) Transitivity: \( (x \equiv y) \Rightarrow (y \equiv z) \Rightarrow (x \equiv z) \)

b. Moreover, it satisfies Leibniz’s principle, here formulated as an axiom

* (L\(\equiv\)) Leibniz: \( (x \equiv y) \Rightarrow \langle p \rangle \equiv \langle p \rangle \)

To exploit this powerful property, we use the axioms for \( \equiv \) for writing earlier properties of \( \Rightarrow \) that come as pairs \( \langle p \Rightarrow q \rangle \) and \( q \Rightarrow p \) in equational form.

(1.35) **Theorem**: **(Earlier theorems in equational form)**

* (ESH\(\Rightarrow\)) Shunting with \( \Rightarrow \): \( x \Rightarrow y \Rightarrow z \equiv y \Rightarrow x \Rightarrow z \)
* (ECP\(\Rightarrow\)) Contrapositive: \( (x \Rightarrow y) \equiv (\neg y \Rightarrow \neg x) \)
* (LE\(\Rightarrow\)) Left identity for \( \Rightarrow \): \( 1 \Rightarrow x \equiv x \)
* (RN\(\Rightarrow\)) Right negator for \( \Rightarrow \): \( x \Rightarrow 0 \equiv \neg x \)
* (E\(\equiv\)) Identity for \( \equiv \): \( (1 \equiv x) \equiv x \)
* (N\(\equiv\)) Negator for \( \equiv \): \( (0 \equiv x) \equiv \neg x \)
* (DNE) Double negation (equationally): \( \neg \neg x \equiv x \)

C. **Additional theorems with \( \equiv \)**

Although case analysis can handle specific propositional problems as they arise, several additional properties are worth listing explicitly because of their general usefulness in calculation. We start by relating equivalence to negation.
(1.36) **Theorem:** (SD$\!\!$/=$\equiv$) Semidistributivity of $\neg$ over $\equiv$

$$\neg (x \equiv y) \equiv \neg (x \equiv y)$$

**Proof** Prove $\neg (x \equiv 0) \equiv \neg (x \equiv 0)$ and $\neg (x \equiv 1) \equiv \neg (x \equiv 1)$, i.e., case analysis. Comparison with a ‘regular’ proof is an instructive exercise.

(1.37) **Theorem:** (A$\!\!$/=$\equiv$) Associativity of $\equiv$

$$((x \equiv y) \equiv z) \equiv (x \equiv (y \equiv z))$$

Associativity allows omitting parentheses by the usual convention. We shall do so henceforth, starting with another little-known variant of Shannon expansion.

(1.38) **Theorem:** Shannon by equivalence

$$p \equiv \neg x \Rightarrow p_0 \equiv x \Rightarrow p_1$$

(1.39) **Theorem:** (LD$\!\!/=$/$\equiv$) Left distributivity of $\Rightarrow/\equiv$

$$z \Rightarrow (x \equiv y) \Rightarrow z \Rightarrow x \equiv z \Rightarrow y$$

Sloppy thinking suggests $(x \equiv y) \Rightarrow z \equiv x \Rightarrow z \equiv y \Rightarrow z$. Case analysis with $z := 1$ reveals no error, but $z := 0$ yields $\neg (x \equiv y) \equiv \neg x \equiv \neg y$, which is in conflict with several earlier theorems, most obviously with (SD$\!\!$/=$\equiv$). The latter theorem directly supplies the patch: negate either $x$ or $y$, but not both. The case $z := 1$ remains unaffected by this patch. So we obtain the following theorem.

(1.40) **Theorem:** (RSD$\!\!/=$/$\equiv$) Right skew distributivity of $\Rightarrow/\equiv$

$$(x \equiv y) \Rightarrow z \equiv x \Rightarrow z \equiv \neg y \Rightarrow z$$

Observe that, due to the associativity and commutativity of $\equiv$, each theorem of the form $p \equiv q \equiv r$ expresses a number of properties with different flavor, e.g., $x \Rightarrow z \equiv \neg y \Rightarrow z \equiv (x \equiv y) \Rightarrow z$. Grouping and weakening from $\equiv$ to $\Rightarrow$ yields even other variants, such as $(x \Rightarrow z) \Rightarrow (\neg y \Rightarrow z) \Rightarrow (x \equiv y) \Rightarrow z$.

**D. Propositional Inequality “$\neq$”**

(1.41) **Axiom for Propositional Inequality**

$$(x \neq y) \equiv \neg (x \equiv y)$$

Via the properties of $\equiv$, especially (SD$\!\!$/=$\equiv$), one quickly deduces the following.

(1.42) **Theorem:** Algebraic laws for $\neq$ (also combined with $\equiv$)

- (IR$\neq$) **Irreflexivity:** $\neg (x \neq x)$
- (S$\neq$) **Symmetry:** $(x \neq y) \equiv (y \neq x)$
- (A$\neq$) **Associativity:** $((x \neq y) \neq z) \equiv (x \neq (y \neq z))$
- (MA$\neq$/=$\equiv$) **Mutual associativity:** $((x \neq y) \equiv z) \equiv (x \neq (y \equiv z))$
- (MI$\neq$/=$\equiv$) **Mutual interchangeability:** $x \neq y \equiv z \equiv x \equiv y \neq z$
1.1.1 Conjunction and disjunction

We introduce two more operators to the calculus, namely the 2-place function symbols “∧” (read “and”) for conjunction, and “∨” (read “or”) for disjunction. These symbols are both given higher precedence than ⇒, but if we want to avoid obscuring duality, it is not a good idea to give to give one of them precedence over the other. In all other respects, this convention reduces the number of parentheses (on the average). Moreover, the chosen precedence rules give the proposition calculus the flavor of an equational algebra, since ≡ and ∧ have the same relative precedence as = and · in arithmetic.

As will become clear from their axioms, these operators do not add structure to the calculus, and hence are mainly for convenience.

A. INTRODUCING CONJUNCTION

Writing the system of implications from Equation 1.30 on page 30 in its simplest form, namely by letting the propositions p and q be just variables, say, x and y, directly yields an implicit definition for this operator.

\[(1.43)\text{ AXIOMS FOR CONJUNCTION ("\(\wedge\")
\begin{itemize}
  \item (AC\(\wedge\)) Accumulation: \[x \Rightarrow y \Rightarrow x \wedge y\]
  \item (W\(\wedge\)) Weakening: \[x \wedge y \Rightarrow x \text{ and } x \wedge y \Rightarrow y\]
\end{itemize}\]

Theorem 1.31 on page 30 provides an equational characterization, which we could have used as an axiom instead, assuming we have ≡ at our disposal.

\[(1.44)\text{ THEOREM: EQUATIONAL DEFINITION FOR "\(\wedge\") } x \wedge y \equiv \neg(x \Rightarrow \neg y)\]

Corollary: Mutual Implication \[(x \equiv y) \equiv (x \Rightarrow y) \wedge (y \Rightarrow x)\]

An equational axiom is often called “explicit” because the new operator appears at only one side of a single equality, whereas the other side contains just ‘old’ operators. Calculationally, this means that such an axiom allows introduction or elimination of the new operator through simple replacement of one side by the other (after instantiation if necessary) using Leibniz’s principle. This is opposed to an implicit characterization by axioms having a non-equational structure or containing the new operator at both sides of the equation.

B. ALGEBRAIC PROPERTIES OF CONJUNCTION

A first collection of properties contains those one would normally check in any algebraic structure. The proofs are left as exercises.

\[(1.45)\text{ THEOREM: BASIC ALGEBRAIC PROPERTIES OF CONJUNCTION}
\begin{itemize}
  \item (E\(\wedge\)) Identity for \(\wedge\): \[x \wedge 1 \equiv x\]
  \item (Z\(\wedge\)) Zero for \(\wedge\): \[x \wedge 0 \equiv 0\]
  \item (IP\(\wedge\)) Idempotency of \(\wedge\): \[x \wedge x \equiv x\]
\end{itemize}\]
• (CD\land) Contradiction: \( x \land \neg x \equiv 0 \)
• (C\land) Commutativity of \( \land \): \( x \land y \equiv y \land x \)
• (A\land) Associativity of \( \land \): \( x \land (y \land z) \equiv (x \land y) \land z \)
• (D\land/\land) Distributivity of \( \land/\land \): \( x \land (y \land z) \equiv (x \land y) \land (x \land z) \)

A second collection of properties pertains to interesting combinations with \( \Rightarrow \).

(1.46) Theorem: Properties of \( \land \) in combination with \( \Rightarrow \)

• (\Rightarrow \to \land) Converting \( \Rightarrow \) to \( \land \): \( x \Rightarrow y \equiv \neg (x \land \neg y) \)
• (\Rightarrow to \land) Yet another variant: \( x \Rightarrow y \equiv x \land y \equiv x \land y \Rightarrow z \)
• (SHA) Shunting from \( \Rightarrow \) to \( \land \): \( x \Rightarrow y \Rightarrow z \equiv x \land y \Rightarrow z \)
• (M\Rightarrow/\land) Monotonicity \( \Rightarrow \) w.r.t. \( \land \): \( (x \Rightarrow y) \Rightarrow x \land z \Rightarrow y \land z \)
• (ABA\land/\Rightarrow) Absorption by \( \land \): \( x \land (x \Rightarrow y) \equiv x \land y \)

\[ \text{However, observe also that} \quad x \land (y \Rightarrow x) \equiv x \]

• (DL\land) Dilemma with \( \land \): \( (x \Rightarrow y) \land \neg (x \Rightarrow y) \Rightarrow y \)
• (LG\land) Leibniz guarded by \( \land \): \( z \land (x \equiv y) \Rightarrow p_{x}^{y} \equiv z \land (x \equiv y) \Rightarrow p_{y}^{y} \)

Many other theorems for implication have the form \( p \Rightarrow q \Rightarrow r \) and, of course, all can be translated by shunting into \( p \land q \Rightarrow r \). We shall not list them here.

A third collection of properties pertains to interesting combinations with \( \equiv \).

(1.47) Theorem: Properties of \( \land \) in combination with \( \equiv \)

• (PD\land/\equiv) Pseudodistributivity of \( \land/\equiv \) (observe the extra term): \( x \land (y \equiv z) \equiv x \land y \equiv x \land z \equiv x \)
• (AB\land/\equiv) Absorption by \( \land \): \( x \land (x \equiv y) \equiv x \land y \)
• \( \neg x \equiv x \land \neg y \)
• (LC\land) Leibniz in context of \( \land \): \( (x \equiv y) \land p_{x}^{y} \equiv (x \equiv y) \land p_{y}^{y} \)
• (SN\land) Shannon negative: \( \neg p \equiv x \land p_{1}^{x} \equiv \neg x \land p_{0}^{x} \)

C. Introducing disjunction

By analogy with the system of implications from Equation 1.30 on page 30, we might ask whether reversing the formulas would yield a viable variant. Caution is necessary, since \( p \Rightarrow q \Rightarrow s \) from Equation 1.30 stands for \( p \Rightarrow (q \Rightarrow s) \) and reversion yields \( (q \Rightarrow s) \Rightarrow p \), whereas we want a variant with antecedent \( s \). The sought variant (whose design is not discussed here) turns out to be

\[ (1.48) \quad p \Rightarrow s \quad q \Rightarrow s \quad s \Rightarrow \neg p \Rightarrow q. \]

This system satisfies the following very unsurprising theorems.

(1.49) Theorem: Existence and uniqueness of solutions to
\[ s \Rightarrow \neg p \Rightarrow q \quad p \Rightarrow s \quad q \Rightarrow s \]
1.1. Introducing additional operators

- Existence: \( s := \neg p \Rightarrow q \) is a solution.
- Uniqueness: if \( s \) and \( s' \) are solutions, then \( s \equiv s' \).

Note also that the pair of implications \( p \Rightarrow s \) and \( q \Rightarrow s \) is interchangeable with the single implication \((\neg p \Rightarrow q) \Rightarrow s\) (exercise).

(1.50) Theorem: Calculation properties of solutions to
\[
\begin{align*}
 s & \Rightarrow \neg p \Rightarrow q & p & \Rightarrow s & q & \Rightarrow s \\
 (r \Rightarrow s) & \Rightarrow r \Rightarrow \neg p \Rightarrow q & (r \Rightarrow \neg p \Rightarrow q) & \Rightarrow r \Rightarrow s
\end{align*}
\]

Hence we are fully justified in proposing the following system of axioms.

(1.51) Axioms for disjunction (“\( \lor \)”)  
- \( x \lor y \Rightarrow \neg x \Rightarrow y \)
- (W\( \lor \)) Weakening to \( \lor \): \( x \Rightarrow x \lor y \) and \( y \Rightarrow x \lor y \)

We obviously also have the following equational characterization.

(1.52) Theorem: Equational definition for “\( \lor \)”  
\( x \lor y \equiv \neg x \Rightarrow y \)

D. Algebraic properties of disjunction

We could derive the list of properties of disjunction independently from those of conjunction. However, duality is too powerful a tool to ignore, and it can be made available immediately by eliminating the implication in the system of equalities \( x \land y \equiv \neg (x \Rightarrow \neg y) \) and \( x \lor y \equiv \neg x \Rightarrow y \), using \( x \equiv \neg^2 x \) where needed.

(1.53) Theorem: (DM) Rules of De Morgan
\[
\neg (x \lor y) \Rightarrow \neg x \land \neg y \quad \neg (x \land y) \Rightarrow \neg x \lor \neg y
\]

The resulting duality principle makes most (if not all) of the following properties direct consequences of the properties for \( \land \) that are their dual.

The issues raised by the first six properties pertain to any algebraic structure.

(1.54) Theorem: Basic algebraic properties of disjunction

- (Ev) Identity for \( \lor \): \( x \lor 0 \equiv x \)
- (Zv) Zero for \( \lor \): \( x \lor 1 \equiv 1 \)
- (IPv) Idempotency of \( \lor \): \( x \lor x \equiv x \)
- (Cv) Commutativity of \( \lor \): \( x \lor y \equiv y \lor x \)
- (Av) Associativity of \( \lor \): \( x \lor (y \lor z) \equiv (x \lor y) \lor z \)
- (DV/\lor) Distributivity of \( \lor / \lor \): \( x \lor (y \lor z) \equiv (x \lor y) \lor (x \lor z) \)
- (CDv) Excluded middle: \( x \lor \neg x \equiv 1 \)
A second collection of properties pertains to interesting combinations with \( \Rightarrow \).

(1.55) **Theorem:** Properties of \( \lor \) in combination with \( \Rightarrow \)

- \(( \Rightarrow \lor \Rightarrow )\) Converting \( \Rightarrow \lor \): \( x \Rightarrow y \equiv \neg x \lor y \)
- \(( \Rightarrow \Rightarrow \lor )\) Yet another variant: \( x \Rightarrow y \equiv x \lor y \equiv y \)
- \((M\Rightarrow/\lor)\) Monotonicity \( w.r.t. \lor: \) \( (x \Rightarrow y) \Rightarrow x \lor z \Rightarrow y \lor z \)
- \((AB\Rightarrow/\lor)\) Absorption by \( \Rightarrow \): \( x \lor (y \Rightarrow x) \equiv y \Rightarrow x \)
  However, observe also that \( x \lor (x \Rightarrow y) \equiv 1 \)

A third collection of properties pertains to interesting combinations with \( \equiv \).

(1.56) **Theorem:** Properties of \( \lor \) in combination with \( \equiv \)

- \((DV/\equiv)\) Distributivity of \( \land/\equiv \): \( x \lor (y \equiv z) \equiv x \lor y \equiv x \lor z \)
- \((AB\lor/\equiv)\) Absorption by \( \lor \): \( x \lor (x \equiv y) \equiv x \lor \neg y \)
- \( x \equiv x \lor y \equiv x \lor \neg y \)
- \((SP\lor)\) Shannon positive: \( p \equiv x \lor p_{1}^{2} \equiv \neg x \lor p_{1}^{2} \)

As expected, our latest guest also has to shake hands with its predecessor.

(1.57) **Theorem:** Properties of \( \lor \) in combination with \( \land \)

- \((GR)\) Golden Rule [14]: \( x \land y \equiv x \lor y \equiv x \equiv y \)
- \((DV/\land)\) Distributivity of \( \lor \) over \( \land \): \( x \lor (y \land z) \equiv (x \lor y) \land (x \lor z) \)
- \((DA/\lor)\) Distributivity of \( \land \) over \( \lor \): \( x \land (y \lor z) \equiv (x \land y) \lor (x \land z) \)
- \((DR\Rightarrow/\lor)\) Right modified distributivity of \( \Rightarrow \) over \( \lor \):
  \( (x \lor y) \Rightarrow z \equiv (x \Rightarrow z) \land (y \Rightarrow z) \)
- \((DR\Rightarrow/\land)\) Right modified distributivity of \( \Rightarrow \) over \( \land \):
  \( (x \land y) \Rightarrow z \equiv (x \Rightarrow z) \lor (y \Rightarrow z) \)
- \((AB\land/\lor)\) Absorption: \( x \lor (x \land y) \equiv x \) and \( x \land (x \lor y) \equiv x \)
- \((ALT\land/\lor)\) Alternative: \( x \lor (\neg x \land y) \equiv x \lor y \) and \( x \land (\neg x \lor y) \equiv x \land y \)
- \( x \equiv x \lor y \equiv x \land y \)
- \((SC\lor)\) Shannon, conventional variants:
  \( p \equiv (\neg x \land p_{1}^{2}) \lor (x \land p_{1}^{2}) \) and \( p \equiv (\neg x \lor p_{1}^{2}) \land (x \lor p_{1}^{2}) \)

To avoid misconceptions, we must warn against the superficial impression that \((DR\Rightarrow/\lor)\) and \((DR\Rightarrow/\land)\) are each other’s dual. Indeed, this impression is definitely false (exercise: explain)! Proofs not using case analysis reveal the different underlying properties (respectively: distributivity and idempotence; exercise).

### 1.2 Remarks on proofs, interpretations and equality

This subsection is meant only for tying up a few loose ends and settling a few issues that did not fit well within the main line of presentation.
1.2. Remarks on proofs, interpretations and equality

1.2.0 Summary of proof techniques

In the development of our calculational variant of proposition logic, we have mentioned some proof techniques that can often considerably shorten deductions. Here we summarize them (including those listed in [14]) with a brief comment.

- **Assuming the antecedent**: to prove \( p \Rightarrow q \), assume \( p \) and deduce \( q \). Warning: \( p \) is normally not a theorem and should not be instantiated during the proof.

- **Case analysis**: to prove \( p \), prove \( p_1 \) and \( p_2 \). Note: choosing \( v \) well can significantly reduce the effort. The method can be used recursively.

- **Proof by negation**: to prove \( p \), prove \( \neg p \Rightarrow 0 \). A classical variant is ‘reductio ad absurdum’: to prove \( \neg p \), prove \( p \Rightarrow 0 \).

- **Proof by contradiction**: to prove \( p \), assume \( \neg p \) and deduce \( p \). Note: the justification is the deduction theorem, \( (\neg p \Rightarrow p) \Rightarrow p \) (exercise), and MP. Warning: this technique also has a degraded ‘junkyard’ version that should be avoided: to prove \( p \), some people assume \( \neg p \) and prove \( p \) without even using the assumption. Many uniqueness proofs fall into this category. Introducing assumptions that remain unused creates an intellectual junkyard.

- **Proof by contrapositive**: to prove \( p \Rightarrow q \), prove \( \neg q \Rightarrow \neg p \).

Observe that the justification of most of these proof techniques directly supplies a mechanical procedure for transforming a proof in which the technique was used into a ‘direct’ proof. The direct proof obtained in this mechanical may be longer, but may also indicate the way to shorter proofs.

1.2.1 Nonstandard interpretations of propositional formulas

One can devise many useful nonstandard interpretations. One example is interpreting the operators of proposition calculus on tuples of binary values by bitwise application of their usual interpretation. Fig. 1.0 illustrates this in the following setting: assuming \( V := \{x, y\} \) to be the set of variables, we took 4-tuples of values in \( \mathbb{B} \) as the domain of interpretation. We chose a state where \( x \) takes value ‘1010’ and \( y \) takes value ‘1100’, which generates the complete 4-cube. Elaboration of the details is left as an exercise. Can you give meaning to the concepts of axiom, inference rule, deduction and theorem with this interpretation?

1.2.2 Remarks on chaining calculation steps

By Leibniz’s principle, in proposition calculus \( (x \equiv y) \Rightarrow (z \Rightarrow x \equiv z \Rightarrow y) \) and \( (x \equiv y) \Rightarrow (x \Rightarrow z \equiv y \Rightarrow z) \). Hence, in chained calculations, we can mix \( \equiv \) with \( \Rightarrow \) as in the following deductions (assuming the propositions between \( (\_\_) \)).

\[
\begin{align*}
r & \Rightarrow (r \Rightarrow p) \\
   & \equiv (p \equiv q) \\
   & \Rightarrow (q \Rightarrow s) \\
   & s
\end{align*}
\]
This is entirely similar to using Leibniz’s principle in arithmetic when writing

\[
\begin{align*}
\langle f \leq d \rangle & \quad d = \langle d = e \rangle \quad e \\
\langle f \leq d \rangle & \quad d = \langle d = e \rangle 
\end{align*}
\]

A general warning is useful here. By convention, when an operator is used as a calculation link —which is recognizable by the presence of \( \langle \rangle \) containing the justification—, it is given the lowest precedence possible, again to reduce parentheses. However, the risk for making mistakes is even greater when working in the lambda calculus. In particular, it can lead to erroneous calculation steps, for instance, assuming \( p \Rightarrow q \Rightarrow r \) is given, the step \( p \Rightarrow q \Rightarrow (p \Rightarrow q \Rightarrow r) \Rightarrow r \). Putting optional parentheses in their proper places can protect against such errors.

### 1.2.3 From propositional equality to equality in general

In the preceding chapters, we formulated some of the laws for equality as inference rules, writing symmetry, transitivity and Leibniz’s principle respectively as

\[
\begin{align*}
\frac{e = e'}{e = e'}, \quad \frac{e = e', e' = e''}{e = e''}, \quad \text{and} \quad \frac{d' = d''}{d[v := d'] = d[v := d'']}
\end{align*}
\]

Here \( e, e', e'' \) and \( d \) are metavariables standing for arbitrary expressions (of types compatible with the operators), and hence these rules can be instantiated in some sense at the metalevel by substituting actual expressions for these metavariables. This is a convenient formulation in a framework without proposition calculus.

However, with proposition calculus at our disposal, we can replace the inference rules by implications in the form of the following axioms.
1.3. Binary algebra

\begin{align*}
(1.58) \text{Axioms: The laws for equality expressed as implications} \\
\quad & \text{Reflexivity: } x = x \\
\quad & \text{Symmetry: } x = y \Rightarrow y = x \\
\quad & \text{Transitivity: } x = y \Rightarrow y = z \Rightarrow x = z \\
\quad & \text{Leibniz: } x = y \Rightarrow e^{x}_{y} = e^{z}_{y}
\end{align*}

These laws hold for variables and expressions of any type (not just propositions).

Within the proposition calculus, \( \equiv \) fulfils the role of equality. We “export” propositional expressions into the remainder of mathematics by the axiom

\begin{align*}
(1.59) \text{Axiom: equivalence as equality} \quad (x = y) \equiv x \equiv y
\end{align*}

Since the variables in this axiom appear as arguments of \( \equiv \), they are propositional, i.e., only propositions should be substituted for them.

1.3 Binary algebra

A \textit{concrete algebra} is a structure consisting of a specific set (e.g., \( \mathbb{Z}, \mathbb{B} \)) together with operators (e.g. +, \( \land \)). By contrast, an \textit{abstract algebra} captures the common properties of concrete algebras, as in group theory and boolean algebra.

Binary algebra is the concrete algebra of operators on a binary set, i.e., a set containing exactly 2 values. The choice of these elements often causes some controversy. Our main criterion is covering the widest possible range of applications in the most convenient way, with emphasis on useful calculational properties. Here we explain the advantages of identifying truth values with the natural numbers 0 and 1, rather than using separate constants like \( \text{F} \) and \( \text{T} \).

A related topic are \textit{conditional expressions}. The aforementioned choice allows to formulate conditional expressions as indexing in a 2-element list, and to derive all required calculation rules in a direct way.

1.3.0 Binary algebra as arithmetic restricted to 0 and 1

Binary algebra can be seen as a model for the abstract algebra of proposition calculus. As such, it offers no reason for preferring any particular choice of truth values. The differences emerge only in the wider context of everyday mathematics.

Various authors \cite{[6, 17]} have shown the calculational advantages of embedding binary algebra into number arithmetic. The common element consists in identifying the binary operators \( \lor \) and \( \land \) with the \textit{least upper bound} \( \lor \) and \textit{greatest lower bound} \( \land \) on numbers.

Informally speaking, we consider the set \( \mathbb{B} := \{0, 1\} \) of values for the binary algebra as a subset of \( \mathbb{R} := \mathbb{R} \cup \{-\infty, +\infty\} \), i.e., the set of real numbers extended with infinity. Ideally, our formal treatment should deal with sets, subsets, algebras, subalgebras etc., but the necessary framework is given only in Chapter 2. Instead, consider an expression language with arithmetic expressions, relational
operators (building atomic propositions) and propositional (boolean) expressions. The formal syntax is not of central importance here, and hence left as an exercise.

Let us define \( \Uparrow \) and \( \Lambda \) for arithmetic \( a \) and \( b \) by

\[
\begin{align*}
(1.60) & \quad a \Uparrow b = (b \leq a) \uparrow a \downarrow b \\
(1.61) & \quad a \Lambda b = (a \leq b) \downarrow a \uparrow b.
\end{align*}
\]

The associated concrete algebra will be called minimax algebra. The conditional expression of the form \( c ? e \downarrow e' \) (for boolean \( c \)) will be taken up again later. For the time being, just take \( (0 ? e \downarrow e') = e' \) and \( (1 ? e \downarrow e') = e \) as axioms.

We often use an equivalent, implicit definition for \( \Uparrow \) and \( \Lambda \), namely

\[
\begin{align*}
(1.62) & \quad a \Uparrow b \leq c \equiv a \leq c \land b \leq c \\
(1.63) & \quad c \leq a \Lambda b \equiv c \leq a \land c \leq b,
\end{align*}
\]

which has certain algebraic and calculational advantages (not elaborated here).

We can define binary algebra as the restriction of minimax algebra to the binary (2-valued) set \( \mathbb{B} \), and rewrite (1.62) and (1.63) as arithmetic equalities:

\[
\begin{align*}
(a \Uparrow b \leq c) & \equiv (a \leq c) \Lambda (b \leq c) \\
(c \leq a \Lambda b) & \equiv (c \leq a) \Lambda (c \leq b).
\end{align*}
\]

In this context, we define logical operators as restrictions of operators on \( \mathbb{R}' \) to \( \mathbb{B} \), more specifically: \( \land, \lor, \equiv \) and \( \Rightarrow \) are restrictions of \( \Lambda, \Uparrow, = \) and \( \leq \) respectively to \( \mathbb{B} \). All laws of minimax algebra thereby particularize to laws over \( \mathbb{B} \), for instance

\[
\begin{align*}
(1.64) & \quad a \lor b \Rightarrow c \equiv (a \Rightarrow c) \land (b \Rightarrow c) \\
(1.65) & \quad c \Rightarrow a \land b \equiv (c \Rightarrow a) \land (c \Rightarrow b).
\end{align*}
\]

The following combined truth table summarizes this particularization for the 16 dyadic functions \( f_i : \mathbb{B}^2 \to \mathbb{B} \) for \( i : 0 \ldots 15 \). To keep matters specific, we made the choice \( \mathbb{B} := \{0, 1\} \). The value of \( f_i (x, y) \) appears in column \( i \) and row \( (x, y) \).

<table>
<thead>
<tr>
<th>( x, y )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0,1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

| \( \mathbb{B} \) |  \forall  |  <  |  >  |  \neq  |  \land  |  \equiv  |  \Rightarrow  |  \Leftarrow  |  \Uparrow  |
| \( \mathbb{R}' \) |  <  |  >  |  \neq  |  \land  |  \equiv  |  \Rightarrow  |  \leq  |  \geq  |  \Uparrow  |

Apart from this embedding in arithmetic, the decision to use numbers instead of distinguished constants entails many other mathematical advantages, e.g.,

a. Applications where the interpretation is more neutral than truth values.

b. Eliminating the need for so-called characteristic functions like “\( C_P = 1 \) if \( P \) holds, 0 otherwise”, often used when calculations become unwieldy.
1.3. Binary algebra

c. Embedding binary algebra within fuzzy logic (\{0, 1\} as a subset of [0, 1]);

d. Simple and direct relation between logical and arithmetical operators, as in

\[ \neg a = 1 - a \quad (a \land b) = a \cdot b \quad (a \lor b) = a + b - a \cdot b. \]

e. Identifying binary algebra with the ring of integers modulo 2, as in

\[ \neg a = 1 \oplus a \quad (a \land b) = a \cdot b \quad (a \lor b) = a \oplus b \oplus a \cdot b. \]

1.3.1 Conditionals as binary indexing

A very convenient framework for calculating with conditional expression is obtained by combining the following three elements.

a. Defining tuples as functions taking natural numbers as arguments in the sense that, for instance \((a, b, c)\) \(0 = a\) and \((a, b, c)\) \(1 = b\) and \((a, b, c)\) \(2 = c\).

b. Embedding the the propositional in arithmetic by treating the basic constants 0 and 1 as numbers.

c. Generic functionals, in particular function composition \((\circ)\) and transposition \((-T)\), formally defined by \((f \circ g) x = f (g x)\) and \(f^T y x = f x y\), without worrying about types for the time being. Observe the analogy with the lambda combinators \(\mathbf{C}\) and \(\mathbf{T}\).

Conditional expressions are written \(c \circ b \downarrow a\). The original formulation is based on two operators \(\circ\) and \(\downarrow\) used in modelling certain analog circuits. However, for our current purpose, it suffices that, for any binary \(c\) and any \(a\) and \(b\),

\[ (1.66) \]

\[ c \circ b \downarrow a = (a, b) c. \]

This combines very conveniently with the distributivity laws for \(-T\) and \(\circ\):

\[ (f, g, h)^T x = f x, g x, h x \quad \text{and} \quad f \circ (x, y, z) = f x, f y, f z. \]

For instance, simple calculation yields two distributivity laws for conditionals:

\[ (1.67) \]

\[ (c \circ f \downarrow g) x = c \circ f x \downarrow g x \quad \text{and} \quad f (c \circ x \downarrow y) = c \circ f x \downarrow f y. \]

Since these calculations are quite similar, we show only one:

\[ (c \circ f \downarrow g) x = \quad \text{(Def. conditional)} \quad (g, f) c x \]

\[ = \quad \text{(Def. transposition)} \quad (g, f)^T x c \]

\[ = \quad \text{(Distributivity \(-T\)} \quad (g x, f x) c \]

\[ = \quad \text{(Def. conditional)} \quad c \circ f x \downarrow g x. \]

In the particular case where \(a\) and \(b\) (and, of course, \(c\)) are all binary, we obtain

\[ (1.68) \]

\[ c \circ b \downarrow a \equiv (c \Rightarrow b) \land (\neg c \Rightarrow a). \]
which is calculationally derived as follows.

\[
\begin{align*}
c ? b \downarrow a & \equiv \ \text{(Def. cond.)} \quad (a, b) c \\
& \equiv \ \text{(Shannon)} \quad (c \land (a, b) 1) \lor (\neg c \land (a, b) 0) \\
& \equiv \ \text{(Def. tuples)} \quad (c \land b) \lor (\neg c \land a) \\
& \equiv \ \text{(Binary alg.)} \quad (\neg c \lor b) \land (c \lor a) \\
& \equiv \ \text{(Defin. \ \Rightarrow)} \quad (c \Rightarrow b) \land (\neg c \Rightarrow a)
\end{align*}
\]

Finally, since predicates are functions and \((z =)\) is a predicate,

(1.69) \[
\boxed{z = (c ? x \downarrow y) \equiv (c \Rightarrow z = x) \land (\neg c \Rightarrow z = y)}
\]

which is calculationally derived as follows.

\[
\begin{align*}
z = (c ? x \downarrow y) & \equiv \ \text{(Distributivity)} \quad c ? (z = x) \downarrow (z = y) \\
& \equiv \ \text{(Preceding law)} \quad (c \Rightarrow z = x) \land (\neg c \Rightarrow z = y)
\end{align*}
\]

These laws are all one ever needs for working with conditionals!
Chapter 2

Calculating with Sets and Functions

The predicate calculus to be introduced in Chapter 3 is *functional* in the sense that predicates are defined as functions, and quantifiers as predicates (hence functions) over predicates. Functions are fully characterized by their domain and their mapping. The domain is the set of values for which the function is specified, and hence is part of the type of the function. In our declarative framework, as in mathematics, it is natural to identify types with sets. We must warn that this form of typing should not be identified in turn with types in programming languages: the latter form is much coarser since it is usually required that type correctness be decidable automatically by a compiler.

In any case, we need minimal rules for calculating with sets and functions. Such rules are presented in this chapter. We obviously assume quantification is not yet available at this stage. This is instructive for exploring how far one can go without quantification, and appreciating why quantification is still necessary.

2.0 Calculating with sets

2.0.0 Set membership, Leibniz’s principle and extensionality

We assume familiarity with sets at the basic level. Here we briefly recapitulate some concepts in a more formal way (without delving into axiomatic set theory) for later use. As usual, we assume the syntax of simple expressions.

The primitive operator is *set membership* (\(\in\)), a relational infix operator. Its left argument can be any expression (introduced gradually throughout these chapters) and its right argument normally a set expression (introduced in this section). Examples are \((p \land q) \in \mathbb{B}\) and \(e/\pi \in \mathbb{R}\), where \(\mathbb{B}\) and \(\mathbb{R}\) are typical constants that, by definition, are set expressions. Applications of \(\in\) are themselves of type \(\mathbb{B}\), as illustrated in \((p \land (e/\pi \in \mathbb{R})) \in \mathbb{B}\).

Let us now characterize *equality* for sets. We adopt Leibniz’s principle

\[
\text{(2.0) Axiom: Leibniz’s principle } \quad e = e' \Rightarrow d^i_e = d^i_{e'}
\]
as a universal axiom for all kinds of expressions \(e, e'\) and \(d\), with the understanding that for abstractions the substitution rules are those of the lambda calculus.

Leibniz’s principle is nothing more than what is to be expected from any well-designed notation. Conventions that can violate it must be considered an abuse of notation and will be avoided. This is neither restrictive, nor hard to do.

For set theory with basic operator \(\in\), Leibniz’s principle yields

\[
x = y \Rightarrow (x \in X \equiv y \in X) \quad \text{and} \quad X = Y \Rightarrow (x \in X \equiv x \in Y).
\]

Obviously, for characterizing set equality, the second part is the important one.

The nontrivial part of the equality is the converse of Leibniz’s principle, which expresses the fact that the basic operators considered indeed constitute a complete characterization of the objects of interest. This amounts to what is usually called extensionality. In other words, Leibniz’s principle expresses what to expect from equality, extensionality expresses the proof obligation to ensure this expectation.

Extensionality cannot be written as \((x \in X \equiv x \in Y) \Rightarrow X = Y\) for reasons that will become clear later. However, it can be expressed as an inference rule.

\[
(2.1) \text{INFECTION RULE: SET EXTENSIONALITY} \quad \frac{q \Rightarrow (s \in S \equiv s \in T)}{q \Rightarrow S = T}
\]

This is a strict inference rule, and also assumes \(s\) is not free in \(q, S, T\).

The role of \(q\) is proof-technical, viz., the deduction theorem and chaining calculations as \(q \Rightarrow \langle \text{Calculations} \rangle (s \in S \equiv s \in T) \Rightarrow \langle \text{Extensionality} \rangle S = T\).

Warning: such a proof is for \(q \Rightarrow S = T\), clearly not \((s \in S \equiv s \in T) \Rightarrow S = T\).

2.0.1 Basic set operators and their axioms

Atomic set expressions are variables declared as such (usually upper case letters like \(X, Y\)) and constants. Examples of constants are the usual symbols for number sets: \(\mathbb{B}\) (binary), \(\mathbb{N}\) (natural), \(\mathbb{Z}\) (integer), \(\mathbb{Q}\) (rational), \(\mathbb{R}\) (real), \(\mathbb{C}\) (complex).

We define the basic constants and operators by axioms reducing all calculations to proposition calculus. As usual, \(s, x, y\) are arbitrary variables, \(d, e\) arbitrary expressions, \(S, T\) set expressions. We also use primes, as in \(e'\).

The constant \(\emptyset\) denotes the empty set, characterized as follows.

\[
(2.2) \text{AXIOM, THE EMPTY SET } \emptyset: \quad x \notin \emptyset \quad \text{(or, by instantiation: } e \notin \emptyset)\]

A set containing exactly the element [denoted by] \(e\) is written \(\iota\ e\). The axiom is

\[
(2.3) \text{AXIOM, THE SINGLETON SET INJECTOR } \iota: \quad x \in \iota\ e \equiv x = e
\]

To denote singletons, \(\iota\ e\) is preferable over \(\{e\}\) for various reasons [10] that will become clear later on. Then we shall also introduce a function range operator \(\{-\}\), which captures common set notations like \(\{a, b, c\}\) and \(\{s : S \mid p\}\) in a clean functional way. Its axiomatization requires quantifiers, but we will see later that

\[
(2.4) \quad e \in \{s : S \mid p\} \quad \equiv \quad e \in S \land p|^e_{e} \quad \text{provided } s \notin \varphi S
\]

which suffices for now. This concludes the operators for denoting sets directly.

Next, we present the usual operators such as \(\cup\) (set union), \(\cap\) (set intersection) which combine sets and thereby establish a boolean algebra of sets.
2.1 Calculating with functions

2.1.0 Function definition

A function is a mathematical object that is fully defined by two attributes: its domain, defining the set of argument “values” (semantically speaking) in which we are interested, and its mapping, defining the properties of an application to arguments in the domain.

For a function \( f \), these two attributes can be axiomatized separately by

- a domain axiom of the form \( Df = S \) or \( x \in Df \equiv p \);
- a mapping axiom of the form \( x \in Df \Rightarrow q \).

where typically \( x \) occurs free in both \( p \) and \( q \), and \( f \) occurs free in \( q \) only.

(2.7) Example The function \( \text{double} \) can be defined by \( D\text{double} = \mathbb{Z} \) together with \( n \in D\text{double} \Rightarrow \text{double} n = 2 \cdot n \).

The function \( \text{halve} \) can be defined by (i) \( D\text{halve} = \{n : \mathbb{Z} \mid n/2 \in \mathbb{Z}\} \) or equivalently, \( n \in D\text{halve} \equiv n \in \mathbb{Z} \land n/2 \in \mathbb{Z} \) for the domain, together with (ii) for the mapping: \( n \in D\text{halve} \Rightarrow \text{halve} n = n/2 \) or, equivalently, \( n \in D\text{halve} \Rightarrow n = \text{double} (\text{halve} n) \).
The two axioms constitute a set of equations with unknown \( f \), for which the
definer must ensure that the solution exists and is unique. In a \textit{specification}, as
opposed to a \textit{definition}, existence and uniqueness are not mandatory.
Uniqueness means that any solutions \( f \) and \( g \) satisfy \( f = g \), where function
equality is addressed soon. It will become clear that these conditions are automat-
ically met if \( x \in \mathcal{D} f \Rightarrow q \) is an \textit{explicit} definition of the form \( x \in \mathcal{D} f \Rightarrow f x = e \),
where \( e \) is an expression not containing \( f \).

(2.8) Example The axiom \( n \in \mathcal{D} \text{halve} \Rightarrow \text{halve} \ n = n/2 \) is an explicit
definition, but \( n \in \mathcal{D} \text{halve} \Rightarrow n = \text{double} (\text{halve} \ n) \) is implicit.

B. DENOTING A FUNCTION BY AN ABSTRACTION

If an explicit mapping definition is available, it can be combined in a succinct
form with the domain information in a single expression denoting the function, obviating even the decision about giving the function a name.

This is achieved by an \textit{abstraction}, namely an expression of the form

\[ v : S \land p . e \quad \text{assuming} \quad v \notin \varphi S \]

where \( v \) is a variable, \( S \) a set expression, \( p \) a proposition and \( e \) any expression.
The part \( \land p \) is optional, and \( v : S . e \) stands for \( v : S \land 1 . e \). Here are the axioms:

(2.9) Axioms for an Abstraction of the form \( v : S \land p . e \)

\begin{itemize}
  \item The domain axiom \( d \in \mathcal{D} (v : S \land p . e) \Leftrightarrow d \in S \land p \uparrow_d \)
  \item The mapping axiom \( d \in \mathcal{D} (v : S \land p . e) \Rightarrow (v : S \land p . e) d = e_d \)
\end{itemize}

The substitution rules are entirely analogous to those of the lambda calculus.

(2.10) Example For \( n \in \mathbb{Z} \land n \geq 0.2 \cdot n \), the domain axiom yields

\[ m \in \mathcal{D} (n : \mathbb{Z} \land n \geq 0.2 \cdot n) \equiv (\text{Domain axiom}) \quad m \in \mathbb{Z} \land (n \geq 0) \uparrow_m \]
\[ \equiv (\text{Substitution}) \quad m \in \mathbb{Z} \land m \geq 0 \]
\[ \equiv (\text{Definition } \mathbb{N}) \quad m \in \mathbb{N} \]

Hence \( \mathcal{D} (n : \mathbb{Z} \land n \geq 0.2 \cdot n) = \mathbb{N} \). If \( x + y \in \mathcal{D} (n : \mathbb{Z} \land n \geq 0.2 \cdot n) \) or,
equivalently, \( x + y \in \mathbb{N} \), the mapping rule yields

\[ (n : \mathbb{Z} \land n \geq 0.2 \cdot n) (x + y) = (\text{Mapping axiom}) \quad (2 \cdot n) \uparrow_{x+y} \]
\[ = (\text{Substitution}) \quad 2 \cdot (x + y) \]

As a consequence, \( \text{double} = n : \mathbb{Z} . 2 \cdot n \) and \( \text{halve} = n : \mathbb{Z} \land n/2 \in \mathbb{Z} . n/2 \).

At first sight, such abstractions look unlike anything encountered in other courses,
including mathematics. In fact, we shall rarely use them in isolation. However,
we will see that, combined with so-called \textit{elastic operators} defined later, they are
ideal for synthesizing traditional notations in a formally correct and more general
way. For instance, \( \sum n : S . n^2 \) will denote the sum of all \( n^2 \) as \( n \) “ranges” over \( S \). What used to be vague intuitive notions will acquire formal calculation rules.
2.1. Calculating with functions

2.1.1 Function equality

A. Function equality: Leibniz’s principle and extensionality

For functions as first-class objects, but in a setting without typing attributes, Leibniz’s principle yields (exercise: make the application of this principle explicit)

\[ x = y \Rightarrow f(x) = f(y) \]
\[ f = g \Rightarrow f(x) = g(x) \]
\[ f = g \Rightarrow \mathcal{D}f = \mathcal{D}g \]

In a strongly typed environment, it is assumed that the arguments are of the correct type for the function. In a more general setting, out-of-domain applications must be tolerated for reasons of generality and flexibility, yet still handled in a useful way. We achieve this by using guards [7], i.e., antecedents in implications to express the domain condition, as in \( x \in \mathcal{D}f \Rightarrow f(x) = e \), which trivially holds when \( x \notin \mathcal{D}f \). This results in the following refinement of the three formulas.

\[ x = y \Rightarrow x \in \mathcal{D}f \land y \in \mathcal{D}f \Rightarrow f(x) = f(y) \]
\[ f = g \Rightarrow x \in \mathcal{D}f \land x \in \mathcal{D}g \Rightarrow f(x) = g(x) \]
\[ f = g \Rightarrow \mathcal{D}f = \mathcal{D}g \]

Only the latter two are relevant to function equality, since the first refers to only one function. We use the definition of set intersection when rewriting the last two rules in combined form and completing it with the converse,

\[
\begin{align*}
(2.11) \text{Axiom and inference rule for function equality} \\
& \bullet \quad (q \Rightarrow f = g) \quad \Rightarrow \\
& \quad q \Rightarrow \mathcal{D}f = \mathcal{D}g \land (x \in \mathcal{D}f \land x \in \mathcal{D}g \Rightarrow f(x) = g(x)) \\
& \bullet \quad \text{Extensionality:} \quad q \Rightarrow \mathcal{D}f = \mathcal{D}g \land (x \in \mathcal{D}f \land x \in \mathcal{D}g \Rightarrow f(x) = g(x)) \quad \Rightarrow \\
& \quad q \Rightarrow f = g
\end{align*}
\]

assuming \( x \) does not occur free in \( q, f, \) or \( g \). The proposition \( q \) is inserted for the same proof-technical reasons as explained for Inference Rule 2.1 on page 44 and also for supporting higher-order functions. Obviously, the inference rule is strict.

B. Specialization to the equality rules for abstractions

With \( f := v : S \land p . d \) and \( g := v : T \land q . e \) (by \( \alpha \)-convertibility, using the same variable \( v \) does not reduce generality), we obtain the equality rules for abstractions.

\[
(2.12) \text{Theorem: equality for abstractions} \\
& \bullet \quad \text{Leibniz:} \quad (v : S \land p . d) = (v : T \land q . e) \Rightarrow \\
& \quad (v \in S \land p \equiv v \in Y \land q) \land (v \in S \land p \Rightarrow d = e) \\
& \bullet \quad \text{Extensionality:} \quad (v \in S \land p \equiv v \in T \land q) \land (v \in S \land p \Rightarrow d = e) \quad \Rightarrow \\
& \quad (v : S \land p . d) = (v : T \land q . e)
\]
To save space and avoid clutter, we omitted the guard and also a redundant $x \in Y \land q$ at the cost of symmetry. Re-inserting these items is a trivial exercise.

Extensionality becomes clearer when factorized into two parts, obtained by expressing it first with $d := e$ and subsequently with $T, q := S, p$.

Extensionality, domain part:
$$v \in S \land p \equiv v \in T \land q \quad \vdash \quad (v : S \land p) = (v : T \land q)$$

Extensionality, mapping part:
$$v \in S \land p \Rightarrow d = e \quad \vdash \quad (v : S \land p, d) = (v : S \land p, e)$$

2.1.2 Generic functionals

A. Design principles

A function or higher order function is a function whose argument or result is a function. We call it generic if it can be used in a wide variety of applications, without restrictions on the arguments.

In mathematics and computing, most functionals are not generic. For instance, with the traditional definition, the function composition operator $\circ$ requires that in an application of the form $f \circ g$ the arguments $f$ and $g$ satisfy $x \in D g \Rightarrow g x \in D f$. If that condition is met, then $f \circ g$ is defined by the domain axiom $D (f \circ g) = D g$ and the mapping axiom $x \in D (f \circ g) \Rightarrow (f \circ g) x = f (g x)$.

We shall redefine such functionals in a generic way by careful bookkeeping of the domains of the arguments when defining the domain of the result.

B. Point-wise and point-free styles

Function definitions and manipulations referring to points in their domains by means of dummies are called point-wise. If no reference is made to points in the domain, the formulation is called point-free.

Traditional mathematical discourse is mostly point-wise. This may be mostly due to the lack of suitable generic functionals to support the point-free style. Yet, practice in many application areas indicates that the point-free style is usually more terse and more elegant (although excesses are better avoided since the point-free style may appear baffling to the uninitiated). Theories about the semantics of programming languages make extensive use of the point-free style.

For practical use, a formalism should support both styles, as well as smooth transformation rules between them. In our formalism, these rules are provided by the axioms defining the functionals.

C. A few generic functionals and their axioms

The archetype rule for transforming between point-wise and point-free forms is the equality $f = x : D f \cdot f x$. The first generic functional, namely filtering, provides a more general form. Many of the other generic functionals typically do not introduce or remove variables, but can move them to other positions where they are more amenable to application of the transformation rules.
2.1. Calculating with functions

Set and function filtering (— ↓ —) A predicate is a function satisfying \( x \in D \implies P x \in B \). For any set \( X \) and predicate \( P \) we define \( X \downarrow P \) by

\[
x \in (X \downarrow P) \iff x \in X \cap D \land P x.
\]

Readers familiar with PVS will notice the similarity with subtyping [23].

Furthermore, for any function \( f \) we define \( f \downarrow P = f | (D \downarrow P) \).

We found that, once available, the restriction operators prove to be useful very frequently, so we abbreviate \( X \downarrow P \) to \( X_P \) and \( f \downarrow P \) to \( f_P \). In addition, we allow partial application for every infix operator \( -- \), i.e. \( a \ast b \) denote functions such that \((a \ast b) b = a \ast b \) and \((b \ast a) = a \ast b \) respectively.

Now we have a formal basis for a variety of notations that are self-explanatory even without the preceding definitions, such as \( Z_{\geq 0} \) and \( R_{\geq 0} \) for sets.

Function composition (— o —) Given functions \( f \) and \( g \), their composition \( f \circ g \) is a function defined as follows, using separate domain and mapping axioms.

\[
x \in D (f \circ g) \iff x \in D g \land g \in D f
\]

\[
x \in D (f \circ g) \implies (f \circ g) x = f (g x).
\]

Equivalently, using abstraction, \( f \circ g = x: D g \land g \in D f \implies f (g x) \). Observe that, if the traditional requirement \( x \in D g \implies g x \in D f \) is satisfied, \( D (f \circ g) = D g \).

Direct extension (— ⊔ —) This operator is inspired by a convention in systems and communications theory, namely extending the arithmetic \(+\) to number-valued functions \( f \) and \( g \) (representing signals in time) by \((f + g) t = f t + g t\).

For generalization to arbitrary operators, we introduce the direct extension operator \( (\sqcup) \). For any 2-argument operator \( -- \), we define \( \hat{\circ} \) such that, given functions \( f \) and \( g \), \( f \hat{\circ} g \) is a function defined as follows.

\[
x \in D (f \hat{\circ} g) \iff x \in D f \cap D g \land (f x, g x) \in D (\ast)
\]

\[
x \in D (f \hat{\circ} g) \implies (f \hat{\circ} g) x = (f x) \ast (g x).
\]

Equivalently, by abstraction, \( f \hat{\circ} g = x: D f \cap D g \land (f x, g x) \in D (\ast) \implies f \hat{\circ} g = x: D f \cap D g \implies f x \ast g x \).

\[(2.13) \text{ Example} \quad \text{Noteworthy is the direct extension of function equality:}
\]

\[
(f \equiv g) = x: D f \cap D g \implies f x = g x, \text{ so } f \equiv g \text{ is a predicate on } D f \cap D g.
\]

Half direct extension (— ⊔ and — ⊔ —) Often we need only half direct extension: for any function \( f \) and any \( x \), we define \( \hat{f} \circ x \) by \( \hat{f} \circ x = f \hat{\circ} D f \cdot x \).

Function override (— ⊗ — and — ⊗ —) For any two functions \( f \) and \( g \), the function \( f \otimes g \) (read "f overriding g") is defined by the axiom pair

\[
D (f \otimes g) = D f \cup D g
\]

\[
x \in D (f \otimes g) \implies (f \otimes g) x = x \in D f \setminus f x \downarrow g x
\]

or, equivalently, by the single axiom \( f \otimes g = x: D f \cup D g \implies x \in D f \setminus f x \downarrow g x \).

Similarly, \( f \otimes g = g \otimes f \) (the operator \( \otimes \) is read "overridden by").
Function merge (\(-\cup-\)) The merge \(f \cup g\) of any two function \(f\) and \(g\) is defined by the following pair of axioms.

\[
\begin{align*}
x \in \mathcal{D}(f \cup g) & \equiv x \in \mathcal{D}f \cup \mathcal{D}g \land (x \in \mathcal{D}f \cap \mathcal{D}g \Rightarrow f x = g x) \\
x \in \mathcal{D}(f \cup g) & \Rightarrow (f \cup g) x = x \in \mathcal{D} f ? f x \downarrow g x
\end{align*}
\]

Function restriction (\(-\upharpoonright-\)) The restriction \(f \upharpoonright X\) of a function \(f\) to a set \(X\) is defined by \(f \upharpoonright X = x : \mathcal{D} f \cap X . f x\) or, equivalently, by

\[
\begin{align*}
\mathcal{D}(f \upharpoonright X) & = \mathcal{D}f \cap X \\
x \in \mathcal{D}(f \upharpoonright X) & \Rightarrow (f \upharpoonright X) x = f x.
\end{align*}
\]

Two relational functionals: compatibility (\(\odot\)) and subfunction (\(\subseteq\)) For any functions \(f\) and \(g\),

\[
\begin{align*}
f \odot g & = f \upharpoonright \mathcal{D} g = g \upharpoonright \mathcal{D} f \\
f \subseteq g & = f = g \upharpoonright \mathcal{D} f.
\end{align*}
\]

All these functionals have interesting algebraic properties that can be derived calculationally. Examples are \(f \subseteq g \equiv \mathcal{D} f \subseteq \mathcal{D} g \land f \odot g\) and the fact that \(\subseteq\) is a partial ordering relation (reflexive, antisymmetric, transitive). Furthermore, \(f \odot g \Rightarrow f \odot g = f \cup g = f \odot g\). Other examples will appear in the exercises.

Constant function definer (\(\langle-\rangle\)), empty function (\(\varepsilon\)) and one-point function definer (\(\langle-\rightarrow-\rangle\)) These are trivial but useful auxiliary functionals.

The basis is \(\langle\rangle\), the constant function definer: \(X \cdot y\) denotes the function with domain \(X\) and (constant) image \(y\). Equivalently, \(X \cdot y = x : X . y\). Example: \(\mathcal{D}(Z \cdot 3) = \mathbb{Z}\) and \(x \in \mathbb{Z} \Rightarrow (Z \cdot 3) x = 3\).

A particular instance is the empty function \(\varepsilon\) defined by \(\varepsilon = 0 \cdot e\) (irrespective of \(e\); exercise: prove \(0 \cdot e = 0 \cdot e'\)).

Another particular instance is the single point function definer \(\rightarrow\). For any \(x\) and \(y\), \(x \rightarrow y = i x \cdot y\). By fortunate coincidence, both the symbol \(\rightarrow\) and its semantics coincide with the maplet in the specification language \(Z\) [24]. However, we must warn that, in AMS publications, \(x \rightarrow e\) stands for \(\lambda x.e\).
Chapter 3

Functional Predicate Calculus

Predicate calculus with quantification is perhaps the most generally applicable formalism throughout pure and applied mathematics, including engineering and computing. Unfortunately, in traditional practice, its true potential has largely remained unexploited, namely its use in calculations. The main culprit is undoubtedly the informal use of \( \exists \) and \( \forall \) as mere abbreviations for “there exists” and “for all”. Another impediment is the way in which the topic is treated in logic textbooks, where the emphasis on proof- and model-theoretic technicalities suggests that the application range is restricted to formal logic.

By contrast, we provide a formal framework with precise calculation rules. We present quantifiers as functional operators, more specifically, as predicates over predicates. Predicates themselves are defined as functions that can take only the values 0 and 1; hence they constitute the simplest possible class of functions imaginable, apart from the entirely trivial (i.e., constant-valued) ones. Moreover, a quantifier like \( \forall \) is an operator that maps a predicate to 1 if it is constant (i.e., trivial) and 1-valued, and to 0 otherwise. The near-triviality of this notion might suggest a paucity of calculation rules, but continuity does not hold here: in fact, the collection of calculation rules turns out to be surprisingly rich.

Calculating with functions and functionals is a routine mathematical activity for most engineering, physics and mathematics students, so our functional formulation of predicates and quantifiers especially benefits from this familiarity.

However, the following presentation does not assume any more background in this respect than high school arithmetic.

3.0 Predicates and quantifiers: basic calculation rules

3.0.0 Introduction: definitions and styles of expression

(3.0) DEFINITION A predicate as a function taking only values 0 and 1.
Formally, a predicate \( P \) is characterized by \( x \in D P \Rightarrow P x = 0 \lor P x = 1. \)

We write \( \text{Pred} \) for the set of well-behaved predicates. Roughly speaking, predicate \( P \) is well-behaved if assuming \( P \in \text{Pred} \) does not cause contradictions\(^{\text{0}}\). We do

\(^{\text{0}}\) Actually we should say “to any of the known contradictions”, since nothing more can be...
not further characterize well-behavedness to leave freedom of choice regarding the
preferred axiomatic set theory [10]. This has no impact on practical use anyway.
That not all predicates are well-behaved is illustrated by the Russell predicate
R := x: P \neg (x x) for which Axiom 2.9 yields R \in P \Rightarrow R R = \neg (R R) and
hence R \notin P. Henceforth, “predicate” stands for “well-behaved predicate”.

We define quantifiers as predicates over predicates.

3.1 Definition: The Quantifiers \( \forall \) and \( \exists \) For any predicate \( P \),

\[
\forall P \equiv P = \mathcal{D} P \cdot 1 \quad \text{and} \quad \exists P \equiv P \neq \mathcal{D} P \cdot 0
\]

We read \( \forall P \) as “everywhere \( P \)” and \( \exists P \) as “somewhere \( P \)”.

This very simple definition (much simpler than the familiar arithmetic \( \Sigma \), which
we formalize later) makes all calculation rules intuitively obvious to any applied
mathematician or engineer, but here we shall outline the axiomatic derivation.

The point-free style is chosen for clarity. Familiar forms like \( \forall x : X . p \) (read
“for all \( x \) in \( X \), [we have] \( p \)”) and \( \exists x : X . p \) (“there exists an \( x \) in \( X \) such that \( p \)”)
are obtained by taking \( x : X . p \) for \( P \) or, with \( x \not\in \varphi \), for \( X \cdot p \). Our attention
to the domains yields some rules that are less common in untyped variants.

The order of derivation avoids forward references. Preceding chapters have
provided sufficient background for safely leaving most of the details as exercises.

3.0.1 Simple consequences of the definitions, duality

Some elementary properties directly follow by “head calculation”.

3.2 Theorem: Simple consequences of the definitions

- For constant predicates: \( \forall (X \cdot 1) \equiv 1 \) and \( \exists (X \cdot 0) \equiv 0 \)
- For the empty predicate: \( \forall \epsilon \equiv 1 \) and \( \exists \epsilon \equiv 0 \)
- For any non-constant predicate \( P \): \( \forall P \equiv 0 \) and \( \exists P \equiv 1 \)

For instance, since \( \forall \) and \( \exists \) are not constant, \( \forall \forall \equiv 0 \) and \( \exists \exists \equiv 1 \).

Illustrative of the algebraic equational style are the following theorems. To the
first of them we also refer as the generalized rule of De Morgan for obvious reasons.
The operator \( \neg \neg \) is the direct extension operator for one-argument functions, and
is (as expected) defined by \( \neg \neg f = g \circ f \) for any functions \( g \) and \( f \).

3.3 Theorem: Duality \( \forall (\neg P) \equiv (\neg \exists) P \)

Proof

\[
\forall (\neg P) \equiv (\text{Def. } \forall (3.1), \mathcal{D} (\neg P) = \mathcal{D} P) \quad \neg P = \mathcal{D} P \cdot 1
\]
\[
\equiv (\neg P = Q \equiv P = \neg Q) \quad P = \neg (\mathcal{D} P \cdot 1)
\]
\[
\equiv (e \in \mathcal{D} g \Rightarrow \mathcal{g} (X \cdot e) = X \cdot (g e)) \quad P = \mathcal{D} P \cdot (\neg 1)
\]
\[
\equiv (\neg 1 = 0, \text{ def. } \exists (3.1)) \quad \neg (\exists P)
\]
\[
\equiv (x \in \mathcal{D} (\mathcal{g} f) \Rightarrow \mathcal{g} f x = g (f x)) \quad \neg \exists P
\]

claimed about the various axiomatizations of set theory.
The proof given here is only one of the many possibilities. The general auxiliary properties given as justification are left as exercises; we advise stating and proving them as lemmata for later use.

(3.4) **Theorem:** Meeting \( \forall P \land \forall Q \Rightarrow \forall (P \land Q) \)

**Proof**

\[
\forall P \land \forall Q \\
\equiv \langle \text{Def. } \forall \rangle \quad P = \mathcal{D} P \cdot 1 \land Q = \mathcal{D} Q \cdot 1 \\
\Rightarrow \langle \text{Leibniz} \rangle \quad \forall (P \land Q) \equiv \forall (\mathcal{D} P \cdot 1 \land \mathcal{D} Q \cdot 1) \\
\equiv \langle \text{Def. } \land \rangle \quad \forall (P \land Q) \equiv \forall x : \mathcal{D} (P \cdot 1) \cap \mathcal{D} (Q \cdot 1) \cdot (\mathcal{D} P \cdot 1) x \land (\mathcal{D} Q \cdot 1) x \\
\equiv \langle \text{Def. } \land \rangle \quad \forall (P \land Q) \equiv \forall x : \mathcal{D} P \cap \mathcal{D} Q . 1 \land 1 \\
\equiv \langle \forall (X \cdot 1) \rangle \quad \forall (P \land Q) \equiv 1
\]

This theorem has a conditional converse: \( \mathcal{D} P = \mathcal{D} Q \Rightarrow \forall (P \land Q) \Rightarrow \forall P \land \forall Q \).

We note in passing that predicates form a Boolean algebra under \( \lor \) and \( \land \) (only partially if the domains differ; exercise!), which is also augmented with the direct extensions of other binary algebra operators (\( \oplus, \equiv, \oplus \) etc.).

The following theorem illustrates how domain aspects typically surface.

(3.5) **Theorem:** Constant Predicates

\[
(a) \quad \forall (X \cdot 0) \equiv X = \emptyset \\
(b) \quad \exists (X \cdot 1) \equiv X \neq \emptyset
\]

**Proof** We were unable to find an equational proof, so we first show implication from left to right.

\[
\forall (X \cdot 0) \equiv \langle \text{Def. } \forall \text{ and } \mathcal{D} (X \cdot e) = X \rangle \\
\equiv \langle \text{Leibniz (Axiom 2.11)} \rangle \\
\equiv \langle x \in X \Rightarrow (X \cdot 0) x = (X \cdot 1) x \rangle \\
\equiv \langle x = X \Rightarrow x = 0 = 1 \rangle \\
\equiv \langle q \Rightarrow 0 \equiv \neg q \rangle \\
\equiv \langle p = 0 \equiv 1, x \notin \emptyset \rangle \\
\equiv \langle \neg p \equiv \neg q \equiv p = q \rangle \\
\equiv \langle \text{set extensionality (2.1)} \rangle \\
\Rightarrow X = \emptyset
\]

The converse is shown using Leibniz’s principle together with \( \forall \varepsilon \equiv 1 \).

The dual form \( \exists (X \cdot 1) \equiv X \neq \emptyset \) is obtained via duality (Theorem 3.3).

### 3.0.2 Case analysis, generalized Shannon, distributivity rules

Recall from proposition calculus the power of case analysis and Shannon expansion. We generalize these to predicate calculus. For any expression \( e \) and variable \( v \), let \( \mathcal{D}_v^e \) denote the intersection of the function domains corresponding to the argument positions where \( v \) occurs in \( e \) (exercise: inductive definition).

(3.6) **Lemma:** Particularization For predicate \( P \), variable \( v \) and expression \( e \) we have \( v \in \mathcal{D}_v^P \land e \in \mathcal{D}_v^P \Rightarrow \forall P^v_v \Rightarrow v = e \Rightarrow \forall P \)
Chapter 3. Functional Predicate Calculus

PROOF
\[ v \in D_P^f \land e \in D_P^r \Rightarrow \langle \text{Leibniz (1.58)} \rangle \quad \Rightarrow \quad v = e \Rightarrow (\forall P)[_v^u \equiv (\forall P)[_v^w]] \]
\[ \Rightarrow \langle \text{Sapp. } \forall[_v^u = \forall, P[_v^u = P] \rangle \quad \Rightarrow \quad v = e \Rightarrow (\forall P \equiv \forall P[_v^w]) \]
\[ \Rightarrow \langle \text{Weakening } \equiv \rangle \quad \Rightarrow \quad v = e \Rightarrow \forall P[_v^w] \Rightarrow \forall P \]
\[ \Rightarrow \langle \text{Shunting} \rangle \quad \Rightarrow \quad \forall P[_v^w] \Rightarrow v = e \Rightarrow \forall P \]

(3.7) **Metatheorem: Case analysis**
\[ \forall P \equiv (\forall \forall P[_v^u] \lor (\neg v \lor \forall P[_v^w])) \]
provided \( B \subseteq D_P^r \) and \( v \in B \).

(3.8) **Metatheorem: Shannon expansion**
\[ \forall P \equiv (v \lor \forall P[_v^u] \land (\neg v \lor \forall P[_v^w])) \]
If \( B \subseteq D_P^r \) and \( v \in B \),

As an application example, we show two similar yet subtly different calculations. Let \( P \) be a predicate and \( x \) a variable not occurring free in \( P \). If \( x \in B \), then
\[ \forall (x \notin P) \equiv \langle \text{Shannon (Thm. 3.8)} \rangle \quad (x \lor \forall (1 \notin P)) \lor (\neg x \lor \forall (0 \notin P)) \]
\[ \equiv \langle \text{Def. } \lor, \text{ rules } \land \rangle \quad (x \lor \forall P) \lor (\neg x \lor \forall (D P \bullet 0)) \]
\[ \equiv \langle \text{Thm. 3.5(a), rules } \land, \lor \rangle \quad (x \lor \forall P) \lor (D P = \emptyset) \]
\[ \forall (x \rightarrow P) \equiv \langle \text{Shannon (Thm. 3.8)} \rangle \quad (x \Rightarrow \forall (1 \rightarrow P)) \land (\neg x \Rightarrow \forall (0 \rightarrow P)) \]
\[ \equiv \langle \text{Def. } \Rightarrow, \text{ rules } \Rightarrow \rangle \quad (x \Rightarrow \forall P) \land (\neg x \Rightarrow \forall (D P \bullet 1)) \]
\[ \equiv \langle \forall (X \bullet 1), \text{ rules } \Rightarrow, \lor \rangle \quad x \Rightarrow P \]

The refinement of the motivations indicated as “rules” is left as an exercise. These and similar calculations prove the following theorem (assuming \( x \notin \varphi P \)):

(3.9) **Theorem: Semidistributivity rules**
If \( x \in B \) then
\[ (a) \text{Semidistributivity } \forall /\land: \forall (x \land \forall P) \equiv (\forall P)[_v^u \lor \exists \exists P[[_v^u]] \]
\[ (b) \text{Right semidistributivity } \forall /\Rightarrow: \forall (x \Rightarrow \exists P) \equiv x \Rightarrow \exists P \]
\[ (c) \text{Left semidistributivity } \forall /\Rightarrow: \exists (P \Rightarrow x) \equiv \exists P \Rightarrow x \]

Of course, theorems (3.6), (3.7) and (3.8) also hold if \( \forall \) is replaced by \( \exists \) (exercise). Hence we obtain in a similar way (or from Theorem 3.9 using duality):

(3.10) **Theorem: Semidistributivity rules**
If \( x \in B \) then
\[ (a) \text{Semidistributivity } \exists /\land: \exists (x \land \exists P) \equiv x \land \exists P \]
\[ (b) \text{Right semidistributivity } \exists /\Rightarrow: \exists (x \Rightarrow \exists P) \equiv (x \Rightarrow \exists P) \land \exists DP \neq \emptyset \]
\[ (c) \text{Left semidistributivity } \exists /\Rightarrow: \exists (P \Rightarrow x) \equiv (\forall P \Rightarrow x) \land \exists DP \neq \emptyset \]

3.1 Instantiation and generalization as metatheorems

In our setting, the usual axioms for quantifiers [20] become (meta)theorems.
3.1. Instantiation and Generalization as Metatheorems

(3.11) **Metatheorem: Instantiation and Generalization**

- **Guarded instantiation:** \((q \Rightarrow \forall P) \Rightarrow q \Rightarrow \exists x \in \mathcal{D} P \Rightarrow P x\)
- **Guarded generalization:** \(q \Rightarrow \exists x \in \mathcal{D} P \Rightarrow P x \quad \leftarrow q \Rightarrow \forall P\)

**Proof**  Instantiate the rules for function equality (Axiom 2.11) with \(f := P\) and \(g := \mathcal{D} P \star 1\). Since \(\mathcal{D} P = \mathcal{D} (\mathcal{D} P \star 1)\) and \(P x = (\mathcal{D} P \star 1) x \equiv P x\), the formula \(\mathcal{D} f = \mathcal{D} g \land (x \in \mathcal{D} f \cap \mathcal{D} g \Rightarrow f x = g x)\) is reduced to \(x \in \mathcal{D} P \Rightarrow P x\). The details are left as a simple exercise.

Incidentally, together with Theorem 3.9(c), our instantiation and generalization theorems happen to be the **functional** and **typed** version of the three axioms typically introduced in various logic textbooks [20] for extending propositional logic to predicate logic. The reader is also invited to verify the deduction theorem.

One can combine instantiation and generalization in the following

(3.12) **Metatheorem, \(\forall\)-introduction/Removal:** Provided \(x \notin \varphi(q, P)\), \(q \Rightarrow \forall P\) is a theorem iff \(q \Rightarrow x \in \mathcal{D} P \Rightarrow P x\) is a theorem.

For the special case \(q = 1\), (3.12) reflects usual practice: to prove \(\forall P\), one proves \(x \in \mathcal{D} P \Rightarrow P x\). With \(\forall (P \Rightarrow q) \equiv \exists P \Rightarrow q\), this special case yields

(3.13) **Metatheorem, Witness:** Provided \(x \notin \varphi(q, P)\), \(\exists P \Rightarrow q\) is a theorem iff \(x \in \mathcal{D} P \Rightarrow P x \Rightarrow q\) is a theorem.

The informal proof scheme reflected by (3.13) is the following: to prove \(\exists P \Rightarrow q\), “take” an \(x\) in \(\mathcal{D} P\) satisfying \(P x\) and prove \(q\).

The general case allows weaving generalization into a calculation chain by a

(3.14) **Convention: Proof schema, Generalizing the Consequent:**

\[
q \Rightarrow (\text{Calculations for } x \in \mathcal{D} P \Rightarrow P x) \quad x \in \mathcal{D} P \Rightarrow P x \\
\Rightarrow (\text{Generalizing the consequent}) \quad \forall P.
\]

The usual warning should be expected by now: this is a proof for \(q \Rightarrow \forall P\), certainly not for \((x \in \mathcal{D} P \Rightarrow P x) \Rightarrow \forall P\). Of course, \(x \notin \varphi(q, P)\).

A first application example is in the proof of the following important theorem.

(3.15) **Theorem: Trading Under \(\forall\)** \(\forall P R \equiv \forall (R \Rightarrow P)\)

**Proof**  First we prove the trading law for \(\forall\) in one direction

\[
\forall P R \Rightarrow \quad \langle \text{Instantiation (3.11)} \rangle \quad x \in \mathcal{D} (P R) \Rightarrow P R x \\
\equiv \quad \langle f R = x : \mathcal{D} f \cap \mathcal{D} R \land R x \Rightarrow f x \rangle \quad x \in \mathcal{D} P \cap \mathcal{D} R \Rightarrow P x \\
\equiv \quad \langle \text{Shunting} \rangle \quad x \in \mathcal{D} P \cap \mathcal{D} R \Rightarrow R x \Rightarrow P x \\
\equiv \quad \langle \text{Axiom } \supset, \text{ remark} \rangle \quad x \in \mathcal{D} (R \Rightarrow P) \Rightarrow (R \Rightarrow P) x \\
\Rightarrow \quad \langle \text{Gen. consequ. (3.14)} \rangle \quad \forall (R \Rightarrow P)
\]

Remark: \(x \in \mathcal{D} P \cap \mathcal{D} R \Rightarrow (R x, P x) \in \mathcal{D} (\Rightarrow)\). The proof for the converse, namely \(\forall (R \Rightarrow P) \Rightarrow \forall P R\), is entirely analogous.
In fact, any proof with this pattern, i.e., starting with instantiation, continuing with equivalences ($\equiv$) only, and concluding with generalization of the consequent, can be reversed in this fashion. We call this the reversion principle.

The trading law for $\forall$ is now used for proving the $\exists$-variant.

(3.16) Theorem: Trading under $\exists$ $\exists P_R \equiv \exists (R \wedge P)$

Proof Using the law for trading under $\forall$, this proof is fully equational.

$$
\exists P_R \equiv \langle \text{Duality (3.3)} \rangle \quad \neg (\forall (\exists P_R))
\equiv \langle \exists P_R = (\exists P)_R \rangle \quad \neg (\forall (\exists P)_R)
\equiv \langle \text{Trading } \forall (3.15) \rangle \quad \neg (\forall (R \Rightarrow \exists P))
\equiv \langle a \Rightarrow b \equiv \neg a \lor b \rangle \quad \neg (\forall (\exists R \wedge \exists P))
\equiv \langle \text{De Morgan} \rangle \quad \neg (\forall (\exists (R \wedge P)))
\equiv \langle \text{Duality (3.3)} \rangle \quad \exists (R \wedge P)
$$

A second application example of schema (3.14) is the proof (exercise) of a characterization of function equality by a single formula without dummies.

(3.17) Theorem: Function equality $f = g \equiv D f = D g \wedge \forall (f \equiv g)$

A final example is the proof (exercise) of the following theorem.

(3.18) Theorem: Merging under $\forall$ $P \odot Q \Rightarrow (\forall (P \wedge Q) \equiv \forall P \wedge \forall Q)$

Observe the similarity with $D P = D Q \Rightarrow (\forall (P \wedge Q) \equiv \forall P \wedge \forall Q)$ obtainable from Theorem 3.4. As usual, there is a dual: $P \odot Q \Rightarrow (\exists (P \cup Q) \Rightarrow \exists P \vee \exists Q)$.

3.2 Completing the predicate calculus

3.2.0 Abstractions synthesizing common notations

Although we do not strictly need abstractions (indeed, our treatment thus far went fine without them), they help synthesizing commonly used notations. For instance, letting $R := x : X . r$ and $P := x : X . p$ in the trading theorems yields

$$
\forall (x : X \wedge r . p) \equiv \forall (x : X . r \Rightarrow p) \quad \text{and} \quad \exists (x : X \wedge r . p) \equiv \exists (x : X . r \wedge p).
$$

For readers not yet accustomed to calculating with direct extensions (although, as the proof for $\exists P_R \equiv \exists (R \wedge P)$ demonstrates, they obey exactly the same formal laws as the operators they extend), we give a direct proof for these formulas instead of presenting them as an instance of the trading theorems.

$$
\forall (x : X \wedge r . p) \Rightarrow \langle \text{Instantiation (3.11)} \rangle \quad x \in D (x : X \wedge r . p) \Rightarrow (x : X \wedge r . p) x
\equiv \langle \text{Axiom 2.9, p. 46} \rangle \quad x \in X \wedge r \Rightarrow p
\equiv \langle \text{Shunting} \rangle \quad x \in X \Rightarrow r \Rightarrow p
\Rightarrow \langle \text{Gen. consequ. (3.14)} \rangle \quad \forall (x : X . r \Rightarrow p)
$$
The converse follows from the reversion principle. As for trading under $\exists$,

$$
\exists(x : X \land r . p) \equiv \langle \text{Duality (3.3)} \rangle \quad \neg (\forall x : X \land r . \neg p)
$$

$$
\equiv \langle \text{Trading $\forall$} \rangle \quad \neg (\forall x : X \land r \Rightarrow \neg p)
$$

$$
\equiv \langle a \Rightarrow b \equiv \neg a \lor b \rangle \quad \neg (\forall (x : X . \neg r \lor \neg p))
$$

$$
\equiv \langle \text{De Morgan} \rangle \quad \neg (\forall x : X . \neg (r \land p))
$$

$$
\equiv \langle \text{Duality (3.3)} \rangle \quad \exists(x : X \land r \land p)
$$

As a further example, we summarize some selected axioms and theorems in tables.

<table>
<thead>
<tr>
<th>Table for $\forall$</th>
<th>General form</th>
<th>Form with $P := x : X . p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>$\forall P \equiv P = \mathcal{D} P \cdot 1$</td>
<td>$\forall (x : X . p) \equiv (x : X . 1)$</td>
</tr>
<tr>
<td>Instantiation</td>
<td>$\forall P \Rightarrow x \in \mathcal{D} P \Rightarrow P x$</td>
<td>$\forall (x : X . p) \Rightarrow x \in X \Rightarrow p$</td>
</tr>
<tr>
<td>Generalization</td>
<td>$x \in \mathcal{D} P \Rightarrow \neg x \lor \exists P$</td>
<td>$x \in X \Rightarrow \neg x \lor \exists (x : X . p)$</td>
</tr>
<tr>
<td>Semidist. $\forall \Rightarrow$</td>
<td>$\forall (q \Rightarrow P) \equiv q \Rightarrow \exists P$</td>
<td>$\forall (x : X . q \Rightarrow p) \equiv q \Rightarrow \forall (x : X . p)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table for $\exists$</th>
<th>General form</th>
<th>Form with $P := x : X . p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>$\exists P \equiv P \neq \mathcal{D} P \cdot 0$</td>
<td>$\exists (x : X . p) \equiv (x : X . 0)$</td>
</tr>
<tr>
<td>$\exists$-introduction</td>
<td>$e \in \mathcal{D} P \Rightarrow P e \Rightarrow \exists P$</td>
<td>$e \in X \Rightarrow p_i^e \Rightarrow \exists (x : X . p)$</td>
</tr>
<tr>
<td>Distrib. $\exists \lor \exists$</td>
<td>$\exists (q \lor P) \equiv q \lor \exists P$</td>
<td>$\exists (x : X . q \land P) \equiv q \lor \exists (x : X . p)$</td>
</tr>
</tbody>
</table>

As a final example, we list some laws for $\forall$ in both styles.

(i) Algebraic style. Let $P$ and $Q$ be predicates and $R$ be a functional with domain $X$ such that $Rx$ is a predicate with domain $Y$ for every $x$ in $X$. Finally, let $S$ be a relation (2-argument predicate).

- **Empty domain** $\forall \varepsilon = 1$
- **One-point rule** $\forall (d \Rightarrow b) = b$ (for boolean $b$)
- **Distributivity** $\mathcal{D} P = \mathcal{D} Q \Rightarrow \forall (P \land Q) = \forall P \land \forall Q$
- **Merge rule** $P \cup Q \Rightarrow \forall (P \lor Q) = \forall P \lor \forall Q$
- **Transposition** $\forall (\forall \circ R^T) = \forall (\forall \circ R^C)$
- **Nesting** $\forall S = \forall (\forall \circ S^C)$
- **Composition rule** $\forall P = \forall (P \circ f) \text{ provided } \mathcal{D} P \subseteq \mathcal{R} f$
- **Trading** $\forall (P \downarrow Q) \equiv \forall (Q \Rightarrow P)$

Here $-^T$ is the transposition operator and $-^C$ the currying operator, both to be defined later, but with the property that $R^T y x = R x y$ and $S^C x y = (S)(x, y)$.

(ii) Using dummies. The selection and the names of the laws come from [9, 14]. Let $p$, $q$ and $r$ be $\mathbb{B}$-valued expressions, and assume evident restrictions on types (which can be made explicit by guards) and free occurrences are satisfied.

- **Empty domain** $\forall (x : X \land 0 . p) = 1$
- **One-point rule** $\forall (x : X \land x = y . p) \equiv y \in X \Rightarrow p_i^y$
- **Distributivity** $\forall (x : X \land r . p) \land \forall (x : X \land r . q) \equiv \forall (x : X \land r . p \land q)$
- **Domain split** $\forall (x : X \land (r \land s . p)) \equiv \forall (x : X \land r . p) \land \forall (x : X \land s . p)$
- **Dummy swap** $\forall (x : X \land r . \forall (y : Y \land q . p)) \equiv \forall (y : Y \land q . (\forall x : X \land r . p))$
- **Nesting** $\forall ((x, y) : X \times Y \land r \land q . p) \equiv \forall (x : X \land r . (\forall y : Y \land q . p))$
- **Dummy change** $\forall (x : X \land r . p) \equiv \forall (y : Y \land r_i^y . p^f_i)$, if $\mathcal{D} f = Y$, $\mathcal{R} f = X$
- **Trading** $\forall (x : X \land q . p) \equiv \forall (x : X . q \Rightarrow p)$
3.2.1 Wrapping up the function package

We introduce some additional generic functionals and a final axiom for functions.

The function range operator $\mathcal{R}$ This operator is defined by the axiom
\begin{equation}
(3.19) \quad y \in \mathcal{R} f \iff \exists (f \equiv y) \quad \text{or, equivalently,} \quad y \in \mathcal{R} f \iff \exists (x : \mathcal{D} f . f x = y).
\end{equation}
Introducing $\{ - \}$ as an operator fully interchangeable with $\mathcal{R}$ synthesizes the common set notations (except for singletons) with their usual form and meaning. First, since lists are functions, expressions like $\{a, b, c\}$ denote a set by listing its elements. Moreover, with the convention (due to Van Thienen [26, 27]) that
\begin{align*}
e | x : X & \quad \text{stands for} \quad x : X . e \\
x : X | p & \quad \text{stands for} \quad x : X \land p . x,
\end{align*}
the expressions $\{e | x : X\}$ and $\{x : X | p\}$ also obtain their usual meaning. For instance, $\{2 \cdot n | n \in \mathbb{Z}\}$ and $\{n : \mathbb{Z} | n/2 \in \mathbb{Z}\}$ both denote the set of even numbers.

Compared with such unification, discarding $\{ - \}$ to denote singleton sets is hardly a sacrifice. It is even an advantage, since using $\{ - \}$ for singletons violates Leibniz’s principle, which requires $x = a, b, c \Rightarrow \{x\} = \{a, b, c\}$. Instead, for singletons, we use a singleton set injection operator $\iota$, defined by $x \in \iota y \equiv x = y$. More arguments are given by Forster [10]. On the other hand, to avoid confusing the uninitiated, we write $\mathcal{R} f$ rather than $\{f\}$ for the range of a function $f$.

The function inverse operator $\leftarrow$ For any function $f$,
\begin{equation}
(3.20) \quad \mathcal{D} f^\leftarrow = \text{Bran} f \quad \land \quad \forall x : \text{Bdom} f . f^\leftarrow (fx) = x
\end{equation}
where bijectivity domain operator $\text{Bdom}$ and the bijectivity range operator $\text{Bran}$ are defined next.
\begin{align}
\text{Bdom} f &= \{x \in \mathcal{D} f | \forall x' : \mathcal{D} f . f x' = f x \Rightarrow x' = x\} \\
\text{Bran} f &= \{fx | x : \text{Bdom} f\}.
\end{align}
Observe that our definition of $f^\leftarrow$ generalizes the usual one by not requiring that $f$ is injective, but coincides with the usual one in case $f$ happens to be injective.

The function arrow operator $\to$ For any sets $X$ and $Y$ and function $f$,
\begin{equation}
(3.23) \quad f \in X \to Y \equiv \mathcal{D} f = X \land \mathcal{R} f \subseteq Y.
\end{equation}

The function comprehension axiom In the examples thus far, we have specified functions by definitions of the form $\mathcal{D} f = X \land x \in \mathcal{D} f \Rightarrow f x = e$ where $e$ does not contain $f$. The equality $f x = e$ is called an explicit image definition. A more general way for introducing functions is based on the following axiom.
\begin{equation}
(3.24) \quad \text{AXIOM} \quad \text{Function comprehension: for any relation } \neg \neg R : X \times Y \to \mathbb{B}, \quad \\
(\forall x : X . \exists y : Y . x R y) \quad \equiv \quad (\exists f : X \to Y . \forall x : X . x R (f x)).
\end{equation}
As a defining property for $x$, the reader can use $x, y \in X \times Y \equiv x \in X \land y \in Y$ for the time being, but an important generalization will be presented later.
3.2. Completing the predicate calculus

3.2.2 Wrapping up the quantifier package

For the sake of comparison, we consider the axioms for the $\sum$-operator: for any \(d\), any numeric \(c\) and any functions \(f\) and \(g\) with finite, nonintersecting domains,

$$\sum \varepsilon = 0 \quad \sum (d \mapsto c) = c \quad \sum (f \cup g) = \sum f + \sum g.$$  

These axioms are called respectively the empty rule, the one-point rule and the merge rule. Of the first and the third, we have already seen analogous rules for quantifiers, namely \(\exists \varepsilon \equiv 0\) and \(\exists (P \cup Q) = \exists P \lor \exists Q\) for compatible predicates.

We now state the one-point rule and the composition rule

\[
y \in \mathbb{B} \Rightarrow \forall (x \mapsto y) = y \\
\mathcal{D} P \subseteq \mathcal{R} f \Rightarrow \forall (P \circ f) = \forall P
\]

which we prove here for \(\forall\) only.

(3.26) **Theorem:** **One-point rule**  \(y \in \mathbb{B} \Rightarrow \forall (x \mapsto y) = y\)

**Proof** Assuming the antecedent \(x \in \mathbb{B}\), we prove the consequent.

\[
\forall (x \mapsto y) \equiv \langle \text{Definition } \mapsto \rangle \ \forall (\iota x \cdot y) \\
\equiv \langle \text{Lemma (3.27)} \rangle \ \iota x = \emptyset \lor y \\
\equiv \langle \iota x \neq \emptyset \rangle \ y
\]

(3.27) **Lemma**  \(x \in \mathbb{B} \Rightarrow (\forall (X \cdot x) \equiv X = \emptyset \lor x)\) (Proof: exercise)

(3.28) **Theorem**  \(\forall P \Rightarrow \forall (P \circ f)\) and \(\mathcal{D} P \subseteq \mathcal{R} f \Rightarrow \forall (P \circ f) \Rightarrow \forall P\)

**Proof** For the first part,

\[
x \in \mathcal{D} (P \circ f) \equiv \langle \text{Definition } \circ \rangle \ x \in \mathcal{D} f \land f x \in \mathcal{D} P \\
\Rightarrow \langle \text{Idempotent } \land \rangle \ x \in \mathcal{D} f \land f x \in \mathcal{D} P \land f x \in \mathcal{D} P \\
\Rightarrow \langle \text{Instantiation} \rangle \ x \in \mathcal{D} f \land f x \in \mathcal{D} P \land (\forall P \Rightarrow P (f x)) \\
\equiv \langle \text{Definition } \circ \rangle \ x \in \mathcal{D} (P \circ f) \land (\forall P \Rightarrow (P \circ f) x) \\
\Rightarrow \langle \text{Weakening} \rangle \ \forall P \Rightarrow (P \circ f) x
\]

Shunning, generalization and distributivity \(\Rightarrow / \forall\) yields \(\forall P \Rightarrow \forall (P \circ f)\).

For the converse, we observe that the antecedent \(\mathcal{D} P \subseteq \mathcal{R} f\) amounts to

\[
\mathcal{D} P \subseteq \mathcal{R} f \equiv \langle \text{Definition } \subseteq \rangle \ \forall y : \mathcal{D} P . y \in \mathcal{R} f \\
\equiv \langle \text{Definition } \mathcal{R} \rangle \ \forall y : \mathcal{D} P . \exists x : \mathcal{D} f . f x = y \\
\equiv \langle \text{Comprehension} \rangle \ \exists h : \mathcal{D} P \rightarrow \mathcal{D} f . \forall y : \mathcal{D} P . f (h y) = y
\]

Assuming this antecedent and letting \(h : \mathcal{D} P \rightarrow \mathcal{D} f\) be a witness,

\[
y \in \mathcal{D} P \equiv \langle \text{Witness } h \rangle \ h y \in \mathcal{D} f \land f (h y) \in \mathcal{D} P \\
\equiv \langle \text{Definition } \circ \rangle \ h y \in \mathcal{D} (P \circ f) \\
\Rightarrow \langle \text{Instantiation} \rangle \ \forall (P \circ f) \Rightarrow (P \circ f) (h y) \\
\equiv \langle \text{Definition } \circ \rangle \ \forall (P \circ f) \Rightarrow P (f (h y)) \\
\equiv \langle f (h y) = y \rangle \ \forall (P \circ f) \Rightarrow P y
\]

Shunning, generalization and distributivity \(\Rightarrow / \forall\) yields \(\forall (P \circ f) \Rightarrow \forall P\).
Corollary  If $\mathcal{D} P \subseteq \mathcal{R} f$ then $\forall (P \circ f) = \forall P$
Chapter 4

Functions, Relations and Induction Principles

Formal rules for calculating with functions in general, and predicates in particular, have been presented and used extensively in earlier chapters. Here we first present the language framework where the seeming variety of expression encountered thus far fits into one single notation consisting of only four constructs. This is achieved by the conceptual step of exploiting the notion of function to its fullest, capturing many kinds of objects that traditionally are not seen as functions. We show how the traditional mathematical conventions can be synthesized in such a way that the usual ambiguities are removed, formal calculation rules become available, and new generalizations are obtained “free of charge”.

Together, the calculation rules and the language constitute the Functional Mathematics formalism, and the concrete version used here is called Funmath.

Next, we define relations as a kind of predicate with tuples as arguments. Since predicates and tuples are both functions, relations directly fit into our functional formalism. We introduce the basic nomenclature about relations used in mathematics, in particular abstract algebra, and more extensively computing science, together with the basic rules for calculating with relations in the point-wise and point-free styles. The latter is especially important in formal semantics.

Finally, a particular kind of relation, called a well-founded relation provides a general basis for all forms of mathematical induction used in everyday mathematical practice, ranging from elementary induction over natural numbers to structural induction over grammatical structures. This provides the necessary basis for all inductive proofs encountered not only in Formele Semantiek and Formele Systeemmodellen but in all other courses in the programme as well.

4.0 Functional Mathematics

4.0.0 Rationale

Functional Mathematics [5, 6] is the approach used throughout this text, whereby mathematical objects are systematically defined as functions whenever possible and convenient, which happens to be the case more frequently than one would
expect at first glance. In fact, the approach turns out to be most fruitful when applied to objects that traditionally were not considered functions (or, at best, something similar but not quite the same). The advantages are easily traced back to the collection of generic functionals that is inherited by all these objects, and replaces the many domain-specific ad hoc conventions and notations. The calculation rules for the generic functions have to be established only once.

### 4.0.1 Using Funmath: a preview

The embodiment of the functional mathematics principle in the concrete syntax that we have been using thus far is called Funmath (Functional mathematics). We briefly point out the similarities and differences with the traditional conventions.

- The similarities reside in the fact that the common conventions can be synthesized by orthogonal combination of the four basic constructs. The expressions written in this way can be easily understood by readers having no prior knowledge about the formalism.

- The differences reside in the elimination of the traditional ambiguities and inconsistencies in notation, the availability of precise formal calculation rules, and extension with new forms of expression and useful shorthands. Since the latter include the point-free style, their use may require from the reader more perspicacity or some prior knowledge about the formalism.

The following example illustrates what we mean by using functions to restructure various notations that are rather amorphous in common practice and possess no systematic calculation rules. We start with the following equality

\[(q^3 - 1)/(q - 1) = 1 + q + q^2 = \sum (1, q, q^2) = \sum i: 0..2.q^i.\]

Observe that all expressions, except \(\sum (1, q, q^2)\), appear familiar, but the functional approach provides an entirely new perspective for nearly all of them.

Traditionally, \(\sum i: 0..2.q^i\) (or \(\sum_{i=0}^2 q^i\)) is read as a so-called sum abstractor \(\sum i: 0..2 (or \sum_{i=0}^2 q^i)\) followed by an expression \(q^i\). Neither of these are functions.

In functional mathematics, \(\sum i: 0..2.q^i\) is read as the application of an elastic operator \(\sum\) to an abstraction \(i: 0..2.q^i\). Both of these denote functions.

Another important functional element of the formalism is that tuples are functions. For instance, \(q^0, q^1, q^2 = i: 0..2.q^i\) since both expressions denote functions with the same domain \(0..2\) and the same mapping, i.e., \((q^0, q^1, q^2) k = (i: 0..2.q^i) k = q^k\) for every \(k\) in \(0..2\). Therefore \(\sum (q^0, q^1, q^2) = \sum (i: 0..2.q^i)\) by Leibniz’s principle.

Equation 4.0 also offers a preview of elastic operators and variadic shorthand.

An elastic operator can be explained\(^0\) as extending a 2-argument operator to any (often infinite) number of arguments, e.g., \(\sum\) extends \(+\) and \(\exists\) extends \(\lor\) (logical or). The argument of an elastic operator is a function as well, systematically grouping the values on which the elastic operator has to “work”. For instance, if \(f\) is a number-valued function whose domain is \(\{a, b, c\}\), then \(\sum f = f a + f b + f c.\)

\(^0\)However, once the principle is established, its ramifications are much wider than shown here.
4.0. Functional Mathematics

The convention of alternating expressions with the same infix operator, such as $1 + q + q^2$ and $a = b = c$, is called variadic shorthand. In Funmath, such notation is always defined as the specialization of an elastic operator to a tuple, e.g., $a + b + c$ is defined as $\sum (a, b, c)$ and $a = b = c$ as $\text{con} (a, b, c)$, where $\text{con}$ is a predicate (on functions) defined such that $\text{con} f$ holds iff $f$ is a constant function.

A small illustration of innovative power of this approach is provided by defining $a \neq b \neq c$ as $\text{inj} (a, b, c)$, where $\text{inj} f$ holds iff $f$ is injective. With this definition, $a \neq b \neq c$ means that not just $a \neq b$ and $b \neq c$ but $a \neq c$ as well, which is precisely what one would like it to mean!

4.0.2 The four syntactic constructs of Funmath

The Funmath language consists of only four syntactic constructs: identifier, (function) application, (function) abstraction and (functional) tuple denotation.

A. IDENTIFIER

An identifier is a pure lexical (typographical) item that can be any alphanumeric name (like $x$, $\text{succ}$, $\text{addl}$) or symbolic name (like $+$, $=$, $\Rightarrow$, $\forall$, $\mathbb{N}$). We assume the symbols of common mathematics and those defined in this text given as constants.

Identifiers can be introduced by a binding. A binding has the form $v : S \land p$ (read “$v$ in $S$ with $p$”), where $v$ is the identifier or a tuple of the identifiers being introduced, $S$ a set and $p$ a proposition. The part $\land p$ is optional, and $v : S$ stands for $v : S \land 1$. Alternative notations are with $p$, standing for $\land p$, and $v := e$ for $v : \{e\}$, where $\{e\}$ denotes the singleton set containing only the object [denoted by] $e$. Occasionally, we shall also use multiple bindings as in $v : S \land p; v' : S' \land p'; v'' : S'' \land p''$ (semicolons as separators).

Note that, in all these forms, the colon (:) clearly marks the place where an identifier is declared. Moreover, in a binding (and there only) the affix convention of an operator is declared, e.g., $\_ f \_ : A \times B \to C$ makes $f$ an infix operator.

A binding alone does not denote anything, but can appear in two contexts [6].

a. One context is the already familiar abstraction, denoting a function. We recall the syntax $\text{binding} \_ \text{expression}$, the equivalent $\text{expression} \_ \text{binding}$ and the variant $\text{binding} \_ \text{proposition}$ discussed in Section 3.2.1 on page 58. The denoted function is defined by Axiom 2.9 on page 46.

The identifiers introduced in the binding of an abstraction are called bound variables or dummies and their scope is local, i.e. they are visible only within the abstraction. Hence their names are unimportant, and may be changed by alpha-conversion, e.g., to avoid name clashes.

b. A definition associates a name with an object. Its general syntax is

$$\text{def} \_ \text{binding}.$$  

The identifiers introduced are called constants and their scope is global, i.e., they are also visible outside the definition.


Formally, a definition \texttt{def} $c : S$ with $p$ declares three items:

(i) the introduction of the identifier $c$ as a global constant;

(ii) the axiom $c \in S \land p$, specifying $c$,

(iii) the existence claim $\exists c : S . p$ and

the uniqueness claim $\forall (c', c'') : S^2 . p_{c'c''} \land p_c \Rightarrow c' = c''$,

or, combined into one formula, $\exists (c : S . p \land \forall c' : S . p_c \Rightarrow c = c')$

In the last item, $c$ is treated as a variable to allow the intended quantification and substitution. The claims are proof obligations for the definer.

(4.1) Example  Consider the following list of candidate definitions.

0. \texttt{def zero} := 0

1. \texttt{def succ} : $\mathbb{N} \rightarrow \mathbb{N}$ with $\forall n : \mathbb{N} . \text{succ} n = n + 1$

2. \texttt{def x} : $\mathbb{N}$ with $x^2 = 4$

3. \texttt{def x} : $\mathbb{N}$ with $x^2 = -4$

4. \texttt{def x} : $\mathbb{C}$ with $x^2 = -4$

5. \texttt{def x} : $\mathbb{Z}$ with $x^2 = 4$

Candidates 0, 1 and 2 satisfy the existence and uniqueness requirements. Number 3 violates the existence requirement, whereas 4 and 5 violate the uniqueness requirement.

A \texttt{where}-expression, with syntax \texttt{expression where binding}, introduces definitions local to \texttt{expression}. The constants in the binding (e.g., in a multiple one) are also visible to each other. An equivalent syntax is \texttt{let binding in expression}.

A \texttt{specification}, with syntax \texttt{spec binding}, is similar to a definition by introducing the constant(s) together with the specifying axiom, but without existence or uniqueness claims. In informal language usage, however, we consider the words “specification” and “definition” interchangeable.

B. Function Application

The default affix convention for function application is \texttt{prefix}, as in $f \mathit{x}$. Beside prefix, we have the familiar \texttt{infix} notation from arithmetic, as in $x + y$, or \texttt{superfix} notation, as in $M^T$. The affix convention of an operator can be overruled by enclosing the operator in parentheses, resulting in the default (prefix) convention, e.g., $x + y = (+) (x, y)$ and $M^T = (T) M$. By convention, we often write $(f \mathit{x}) \mathit{y}$ as $f \mathit{x} \mathit{y}$ and, correspondingly, $A \rightarrow B \rightarrow C$ for $a \rightarrow (B \rightarrow C)$. For instance, given \texttt{def add} : $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ with $\text{add} \ x \ y = x + y$, then clearly $\text{succ} = \text{add} \ 1$.

For every operator, the affix convention is specified by the operator’s designer at the place where the operator is first introduced by its \texttt{binding}.

For a single-argument operator, a dash indicates the argument position, e.g., $-^T$ for the function transposition operator and $\{-\}$ for the function range operator defined later. A tuple is considered a single argument. For that purpose,
the same dash is repeated, as in − + −. Repeated dashes are taken from left to right and from top to bottom. An example with one quadruple argument is

$$\underbrace{\ldots}_{\text{in the specification, indicating that}} \quad (f)\ (a, b, c, d).$$

In such an application, components of the tuple argument can be omitted. The result, called a *partial application* (or section in the functional programming jargon [18, 28]), denotes a function that is applicable to the missing component(s). For instance, the specification − * − leads to \((x \ast y) = x \ast y = (\ast y) x\).

We say that an operator has several arguments in case the application of the operator to an argument again denotes a function (which can be applied to the second argument) and so on. As a variant of Curry’s notation [8], numbered dashes indicate the positions of the successive arguments. For instance, the specification for the default prefix convention for a 2-argument operator is \(f \longrightarrow_1\) but, being the default, it is omitted. A more illustrative 4-argument example is

$$\underbrace{\ldots}_{\text{in the specification, indicating that}} \quad \underbrace{\ldots}_{(f)\ a\ b\ c\ d}.\$$

Dashes with the same number (for parts of the same tuple) are taken from left to right and from top to bottom. If another ordering is desired for the items in a tuple, this is done by a second index. We shall wait with an example until the need arises, but we also strongly advise against overly exotic affix conventions.

C. ABSTRACTION

The general form binding expression, the equivalent expression | binding and the variant binding | proposition are discussed in Section 3.2.1 on page 58, and the denoted function is defined by Axiom 2.9 on page 46.

D. TUPLE DENOTATION

This is the fourth syntactic construct, and the only one not discussed thus far.

(4.2) **Definition: Syntax of tuple denotation** A tuple denotation

is a list of at least 2 expressions\(^1\) separated by commas, e.g., \((e, e', e'')\). As usual, the outer pair of parentheses is optional.

We shall see later how ‘,’ can be seen as an infix operator, although it is a basic delimiter. For convenience, we give the comma lower precedence than arithmetic operators but higher than relational ones. For instance, \(a, b = c, d\) stands for \((a, b) = (c, d)\). Since ‘,’ has lower precedence than a prefix operator, the parentheses are not optional in formulas like \((a, b, c) 2 = c\) and first \((a, b, c) = a\).

(4.3) **Axioms for tuple denotation** These axioms can be expressed in their general form at the meta-level only (not elaborated here). Here we express them by example, with intuitively obvious generalization.

$$\mathcal{D} (a, b) = \mathbb{N}_{<2} \quad \land \quad (a, b) 0 = a \land (a, b) 1 = b$$

$$\mathcal{D} (a, b, c) = \mathbb{N}_{<3} \quad \land \quad (a, b, c) 0 = a \land (a, b, c) 1 = b \land (a, b, c) 2 = c.$$  

\(^1\)The empty tuple and the 1-element tuples are particular instances of concepts that are more general than tuple denotation, and hence will be discussed at the appropriate time.
Tuple denotation is not restricted to one dimension, but includes matrix denotations as well. In principle, we need no additional syntax, since we can define a matrix as a tuple of tuples, for instance \((a, b, c), (d, e, f)\). Yet, for convenience\(^2\), we include the possibility of writing \((a, b, c), (d, e, f)\) as

\[
\begin{pmatrix}
a & b & c \\
d & e & f
\end{pmatrix}.
\]

We briefly illustrate how to calculate with tuple denotations. The basic observation is that \(a, b, c = (0 \mapsto a) \cup (1 \mapsto b) \cup (2 \mapsto c)\) etc. by function equality.

**Example** Let us calculate \(\forall (x, y)\), assuming boolean \(x, y\).

\[
\begin{align*}
\forall (x, y) &= (x \land y = (0 \mapsto x) \cup (1 \mapsto y)) && \forall ((0 \mapsto x) \cup (1 \mapsto y)) \\
&= (\text{Note, } \forall (f \cup g) = \forall f \land \forall g) && \forall (0 \mapsto x) \land \forall (1 \mapsto y) \\
&= \forall (a \mapsto b) = b && x \land y.
\end{align*}
\]

Note: \(D (0 \mapsto x) \cap D (1 \mapsto y) = \emptyset 0 \cap 1 = \emptyset\) ensures compatibility.

Similarly one proves \(\sum (x, y) = x + y\) for numeric \(x, y\), based on the following axioms for \(\sum\): the empty rule \(\sum \epsilon = 0\), the one-point rule \(\sum (d \mapsto c) = c\) (any \(d\), numeric \(c\)) and the merge rule \(\sum (f \cup g) = \sum f + \sum g\) for any numeric functions \(f\) and \(g\) with finite, nonintersecting domains.

Not only tuples are defined as functions, but any kind of sequence as well. The basic concepts and operators are discussed next.

### 4.0.3 Sequences and tuples as functions

Sequences play an important role throughout mathematics and engineering. Most often, they are considered as something different from functions, if not intuitively, then certainly in the style of calculation. The resulting calculation rules are ad hoc or even nonexistent, and calculation is done on the basis of interpretations.

Perhaps most undesirable is the ubiquitous use of ellipsis, namely writing dots (\(\ldots\)) for missing parts in an expression, as in \(a_0, a_1, \ldots, a_n\), for which the reader is supposed to infer the missing parts. This is a clumsy form of expression and, by lacking precise calculation rules, an obstacle to formal manipulation. Indeed, it even violates Leibniz’s principle, as illustrated in the following example.

**Example** Consider the “expression” \(a_0 + a_1 + \ldots + a_n\), given \(a_i = i^2\).

According to Leibniz’s principle, this should equal \(0^2 + 1^2 + \ldots + n^2\) and, with \(n = 7\) (for instance), \(0 + 1 + \ldots + 49\), which is not the intended sum.

Although “repairs” can be found for various particular cases, our definition of tuples and sequences as functions together with generic functionals constitutes a better alternative that obviates ellipsis altogether.

---

\(^2\)An alternative form of a syntactic construct is called syntactic sugar, since it is meant only as a convenience. It is very important to remember that it should never be considered a new construct: its calculation rules are only those, and all those, of the basic form. The first reflex in any calculation should be replacing it (explicitly or implicitly) with the basic form.
A. Function domains for sequences and tuples

We define sequences as well as tuples of size \( n \) as functions with domain \( \square n \). The domain \( \square n \) is specified using the auxiliary block operator \( \square \):

\[
\text{def } \square : \mathbb{Z}^+ \rightarrow \mathcal{P} \mathbb{N} \text{ with } \square n = \{ m : \mathbb{N} \mid m < n \}.
\]

Here we assume the definition \( \mathbb{Z}^+ := \mathbb{Z} \cup \{-\infty, +\infty\} \).

The only distinction between sequences and tuples is that all elements of a sequence are of a same given type, whereas for a tuple the types specified for the elements may be different. This distinction is not essential at this stage, since all elements of a tuple belong to the union of the specified types.

(4.7) Example  Clearly, \( \square n = \emptyset \) for all \( n : \mathbb{Z}_{\leq 1} \), in particular, \( \square 0 = \emptyset \). On the other hand, \( \square 1 = \{ 0 \} \), \( \square 2 = \mathbb{B} \) and \( \square \infty = \mathbb{N} \).

A typical sequence of numbers is defined by any of the following definitions, whose equivalence follows from function equality.

\[
\text{def } a := 1, 2, 4, 8, 16 \\
\text{def } a := i : \square^5 \cdot 2^i \\
\text{def } a : \square^5 \rightarrow \mathbb{N} \text{ with } a = 1, 2, 4, 8, 16 \\
\text{def } a : \square^5 \rightarrow \mathbb{N} \text{ with } a i = 2^i
\]

B. Operators for sequences and tuples

For versatile concepts like sequences and tuples, many useful operators can be defined. Here we present the basic ones, and more will be added as needed.

(4.8) Definitions: Basic operators for tuples and sequences [4]

- The length operator \( (\#) \) with axiom \( \mathcal{D} x = \square (\# x) \) for any tuple \( x \).
- The empty tuple or sequence is \( \varepsilon \).
- The singleton tuple operator \( (\tau) \) with axiom \( \tau x = 0 \mapsto x \) for any \( x \).
- The shift operator \( (\sigma) \) with axioms \( \# (\sigma x) = \# x - 1 \) and \( \sigma x n = x (n + 1) \) for nonempty \( x \).
- The concatenation operator \( (\mathbf{++}) \) is defined for any tuples \( x \) and \( y \) by the domain axiom \( \# (x + y) = \# x + \# y \) and the mapping axiom \( \forall i : \mathcal{D} (x + y) . (x + y) i = (i < \# x) ? x i : y (i - \# x) \).
- The prefixing \( (\mathbf{-->}) \) and postfixing \( (\mathbf{<-}) \) operators with axioms \( a \mathbf{-->} x = \tau a + + x \) and \( x \mathbf{<-} a = x + + \tau a \).

For concatenation, a similar definition with different symbol appears in [15, page 19]. The choice of \( + + \) reflects prevalent use in the algorithmics community [3, 21].

(4.9) Examples  For concatenation: \((3, 4) + + (4, 5) = 3, 4, 4, 5\). For prefixing: \((3) \mathbf{-->} (4, 5) = 3, 4, 5\) and \((3, 4) \mathbf{-->} (4, 5) = (3, 4), 4, 5\).
C. Sequence Types

As expected, sequence types are defined as sets of functions. The basic auxiliary operator is the exponentiation or power ($\uparrow$) extended to sets as follows: for any set $A$ and any $n : \mathbb{Z}'$ we define $A \uparrow n = \square n \to A$. As usual, we write $A^n$ for $A \uparrow n$.

By further allowing the second argument of $\uparrow$ to be the symbol $*$ (known as the Kleene star) or $\omega$, we obtain the following classification of sequence types.

- $A^n$ is the set of sequences of length $n$ over $A$, viz. $A^n = \square n \to A$;
- $A^*$ is the set of finite sequences or lists over $A$, viz. $A^* = \bigcup n : \mathbb{N}. A^n$;
- $A^\omega$ is the set of sequences over $A$, i.e. tuples consisting of elements of $A$.

Clearly $A^* \cap A^\omega = \emptyset$ and $A^* \cup A^\omega = A^\omega$.

D. Tuple Types

In a sequence of type $A \uparrow n$, all elements are of the same specified type $A$. We could use this type for a tuple by taking $A$ to be the union of the types specified for the elements. That is clearly rather coarse: it would be more elegant to reflect type information about the individual elements in the type of the tuple.

The necessary refinement can be achieved by a more general operator introduced next.

4.0.4 The generalized functional Cartesian product

Most common function types have the form $X \to Y$, with uniform range.

Finer typing is provided by an operator designed to formalize the concept of tolerance for functions. Engineering in the analog domain assumes certain tolerances on components. To extend this to functions, we introduce a tolerance function $T$ that specifies, for every value $x$ in its domain, the set $Tx$ of allowable values. More precisely, a function $f$ meets the tolerance $T$ iff

$$\mathcal{D} f = \mathcal{D} T \land x \in \mathcal{D} f \cap \mathcal{D} T \Rightarrow f x \in T x.$$  

The principle is illustrated pictorially in Fig. 4.0, using the example that provided the original motivation, namely a radio frequency filter characteristic.

![Figure 4.0: The function approximation paradigm](image)
4.0. Functional Mathematics

We capture this principle by defining an operator \( \times \): for any family \( T \) of sets,

\[
(4.10) \quad f \in \times T \equiv \mathcal{D} f = \mathcal{D} T \land \forall x : \mathcal{D} f \cap \mathcal{D} T. f x \in T x
\]

We call \( \times \) the generalized functional Cartesian product. It expresses the dependency of the result type on the domain value. If \( \times T \neq \emptyset \), then \( \times^{-1} (\times T) = T \).

To help the reader becoming familiar with this operator and obtaining a first appreciation of its very general nature, we provide various application examples.

First, from the analogy with function equality

\[
f = g \quad \equiv \quad \mathcal{D} f = \mathcal{D} g \land \forall x : \mathcal{D} f \cap \mathcal{D} g. f x = g x,
\]

it is easy to show \( f = g \equiv f \in X (u \circ g) \), i.e., \( \times \) also covers the exact case.

A second instructive exercise is elaborating \( \times (A, B) \) for sets \( A \) and \( B \) (since tuples are functions). This yields \( \times (A, B) = A \times B \), the common Cartesian product defined by \( (a, b) \in A \times B \equiv a \in A \land b \in B \). If \( A \neq \emptyset \) and \( B \neq \emptyset \), then \( \times^{-1} (A \times B) 0 = A \) and \( \times^{-1} (A \times B) 1 = B \).

Thirdly, letting \( T := a : A . B \) with \( a \) free in \( B \), it is easy to see by simple calculation that \( \times (a : A . B) = \{ f : A \to \cup a : A . B \mid \forall a : A . f a \in B \} \). This is known as a dependent type [16]. We write \( A \triangleright a \to B_a \) as a suggestive shorthand for \( \times a : A . B_a \), for instance \( A^+ \triangleright x \to A^\# x^{-1} \) (which nicely characterizes the type of the aforementioned \( \sigma \)-operator). This shorthand is especially useful in chained dependencies, e.g., \( A \triangleright a \to B_a \triangleright b \to C_{a,b} \).

A final example is expressing record types à la Pascal [19]. We let field names be elements of an enumeration type, for instance, \{name, age\}. Then, given

\[
def \text{person} := \times (\text{name} \mapsto \mathbb{A}^* \cup \text{age} \mapsto 0..199)
\]

a declaration such as \( x : \text{person} \) means that \( x \text{name} \in \mathbb{A}^* \) and \( x \text{age} \in 0..199 \).

Encapsulation in a functional record type operator is a simple exercise.

4.0.5 Elastic operators and variadic shorthands

A. ELASTIC OPERATORS: PRINCIPLE, DOMAIN MODULATION

Elastic operators are functionals, usually domain-specific but occasionally generic, that replace ad hoc abstractors, and also support the point-free style.

For instance, the following expressions in traditional notation

\[
\forall x : X . P x \quad \sum_{i=m}^{n} x_i \quad \lim_{x \to a} f x
\]

contain the ad hoc abstractors \( \forall x : X, \sum_{i=m}^{n} \) and \( \lim_{x \to a} \). By introducing suitable operators \( \sum \) and \( \lim \), the three expressions can be formalized by defining them to be equal respectively to expressions like

\[
\forall (x : X . P x) \quad \sum i : m .. n . x_i \quad \lim (x : \mathbb{R}_{<a} . f x) a
\]

\footnote{Do not confuse the three-position macro \( \rightarrow \rightarrow \rightarrow \) with the function arrow \( \to \)!}
which are easily understood by the casual reader, or the point-free expressions

\[ \forall P \sum x \lim_{f \downarrow a} a \]

(assuming \( \mathcal{D} P = X \) and \( \mathcal{D} x = m \ldots n \); otherwise use filtering) for the initiated.

Observe that the “extent” of quantification, summation etc. is not specified by the operator but by the domain of the argument, which is especially convenient since the domain is an essential characteristic of every function. This practice is called domain modulation. Conversion between pointwise and point-free expressions is readily achieved by the equality \( f \downarrow P = x : \mathcal{D} f \cap \mathcal{D} P \wedge P . f x \).

### B. Variadic shorthand

A variadic function is a function with an arbitrary number of arguments. Clearly our elastic operators are variadic over tuples of suitable type.

By variadic shorthand we mean operand/operator alternations, as in \( a + b + c \). The following table is a list of examples with their traditional justifications. It is assumed that \( a, b \) and \( c \) are numbers in the first example, booleans in the second, and arbitrary in the next two examples, whereas \( A, B \) and \( C \) are sets.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Traditional justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a + b + c )</td>
<td>associativity,</td>
</tr>
<tr>
<td>( a \lor b \lor c )</td>
<td>associativity</td>
</tr>
<tr>
<td>( a = b = c )</td>
<td>conjunctionality</td>
</tr>
<tr>
<td>( a \neq b \neq c )</td>
<td>not allowed</td>
</tr>
<tr>
<td>( A \cup B \cup C )</td>
<td>associativity</td>
</tr>
<tr>
<td>( A \times B \times C )</td>
<td>special convention</td>
</tr>
</tbody>
</table>

Associativity for an operator \(-*\) : \( A^2 \rightarrow A \) means \( x * (y * z) = (x * y) * z \) and conjunctionality for a relation \(-R-\) : \( A^2 \rightarrow B \) means \( x R y R z \equiv x R y \wedge y R z \).

Instead of resorting to these various different ad hoc justifications, we uniformly define all variadic shorthands as particularizations of elastic operators to tuples, as shown in the following table.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Funmath justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a + b + c )</td>
<td>( \sum (a, b, c) )</td>
</tr>
<tr>
<td>( a \lor b \lor c )</td>
<td>( \exists (a, b, c) )</td>
</tr>
<tr>
<td>( a = b = c )</td>
<td>( \text{con} (a, b, c) )</td>
</tr>
<tr>
<td>( a \neq b \neq c )</td>
<td>( \text{inj} (a, b, c) )</td>
</tr>
<tr>
<td>( A \cup B \cup C )</td>
<td>( \bigcup (A, B, C) )</td>
</tr>
<tr>
<td>( A \times B \times C )</td>
<td>( \times (A, B, C) )</td>
</tr>
</tbody>
</table>

The constant function and the injective function predicates, are defined by

\begin{align*}
\text{def} \ & \text{con} : \mathcal{F} \rightarrow \mathbb{B} \quad \text{with} \quad \text{con} f \equiv \forall (x, y) : (\mathcal{D} f)^2 . f x = f y \\
\text{def} \ & \text{inj} : \mathcal{F} \rightarrow \mathbb{B} \quad \text{with} \quad \text{inj} f \equiv \forall (x, y) : (\mathcal{D} f)^2 . f x = f y \Rightarrow x = y.
\end{align*}

Observe that \( a \neq b \neq c \equiv a \neq b \lor b \neq c \lor c \neq a \), as one would prefer.
4.0. Functional Mathematics

C. A CONVENIENT AUXILIARY CONCEPT: FAMILIES AND FAMILY TYPES

In the technical and scientific literature, one often encounters the term family, as in ‘a family of sets’, but in an informal and vague way. Closer inspection reveals that, in this usage, the term ‘family’ can be considered a synonym for either a set (whose elements are the items of interest) or a function (mapping elements of some ‘index set’ to the items of interest). We choose the latter option, namely “family” as a synonym for “function” for the sake of uniformity (at the minor cost of being occasionally overspecific).

The difference in flavor or bias between a family (as an indexed collection of items) and a function in general is, at most, that the domain of a family is meant just for bookkeeping (e.g., indexing) and that, therefore, its choice is secondary. Of course, having identified families with functions, we discard this bias in favor of generalized use. This is captured by an operator $\text{fam}$ whereby $\text{fam} \, X$ is the set of functions such that the images of all domain elements are of type $X$.

\begin{equation}
\text{def fam} : \mathcal{T} \to \mathcal{P} \mathcal{F} \text{ with } \text{fam} \, X = \{ f : \mathcal{F} \mid \forall x : D \, f \, x \in X \}. 
\end{equation}

\noindent \textbf{Warning} Just as we read $n \in \mathbb{N}$ as “$n$ is a natural number”, we read $f \in \text{fam} \, \mathbb{N}$ as “$f$ is a family of natural numbers”. Note that, in $f \in \text{fam} \, \mathbb{N}$, the family in question is the function $f$, not the set $\text{fam} \, \mathbb{N}$ containing the families like $f$. This remark may appear trivial, but we have found that students experience systematic difficulties in understanding the distinction.

D. DESIGNING ELASTIC OPERATORS

Elastic operators allow considerable design freedom, but in case it is also used as the basis for variadic shorthand for a commonly used infix operator, care must be taken to ensure compatibility with the common conventions (if that is desirable). For new operators, there is complete freedom, so the only criterion is supporting the algebraic properties of the corresponding infix operator.

\begin{equation}
\text{(4.14) Definition: Elastic Extensions } \text{ We say that an operator } F : D \to B \text{ is an elastic extension of an infix operator } \ast \to : A^2 \to B \text{ iff }
\end{equation}

$$A^\ast \subseteq D \subseteq \text{fam} \, A \text{ and } \forall (a, a') : A^2 : F(a, a') = a \ast a'.$$

For maximum generality, $D = \text{fam} \, A$, but this cannot be achieved in a meaningful way for all $\ast$. Examples where full generality is achieved are the quantifiers where, for instance, $\forall \ast = \text{fam} \, \mathbb{B}$ and

$$\mathbb{B}^\ast \subseteq \text{fam} \, \mathbb{B} \text{ and } \forall (b, b') : \mathbb{B}^2 : \forall (b, b') = b \land b'.$$

Variadic extensions for given $\ast$ are not unique, so design decisions must be made.

The corresponding variadic shorthand may have a justification on the basis of desired algebraic properties, and these must be reflected in the elastic extension.

As an example, we consider associativity. It can be shown that, if an infix operator is associative, then the pairing parentheses may be rearranged for any number of terms, e.g.,

$$(a \ast b) \ast (d \ast c) = a \ast (b \ast ((c \ast d) \ast e)).$$
By Definition 4.14, this associativity is inherited by any elastic extension $F$ for any application to pairs, for instance

$$F(a, F(a', a'')) = F(F(a, a'), a').$$

However, this does not automatically ensure that, for instance,

$$F(a, a', a'') = F(F(a, a'), a').$$

Indeed, a trivial counterexample consists in making $F$ constant for any tuple longer than 2.

The condition that an elastic operator $F$ has to support the associativity convention for the variadic notation for any number of terms is expressed by

$$F(x++y) = F x y$$

for all $x$ and $y$ in $A^\ast$. In other words, $F$ must be a list homomorphism. This condition implies (exercise) that the range of $F$ is a monoid with identity element $F\varepsilon$, and also that there exists $g:A \to B$ such that $F(a \triangleright x) = g a \ast F x$, with $A$ and $B$ obtained from the conventions of Definition 4.14.

The first of the following series of application examples illustrates this issue.

E. SOME TYPICAL DESIGNS

Earlier examples Operators such as $\forall$ and $\times$ that we have already encountered were designed according to the aforementioned principles. Obviously, $\forall$ reflects associativity, but $\times$ emphatically does not.

Summation The $\sum$ operator is defined by the equations

$$\sum \varepsilon = 0 \quad \sum (a \mapsto c) = c \quad \sum (f \cup g) = \sum f + \sum g$$

for any $a$, any numeric $c$, and any number-valued functions $f$ and $g$ with non-intersecting and finite domains. Extension to infinite ordered domains can be defined by the usual limit construction in cases where convergence is assured.

We define Fourier’s notation $\sum_{i=m}^n e$ to be an abbreviation for $\sum (i : m \ldots n : e)$, where the latter “$\sum$” is our elastic operator, and “$\ldots$” is the upto operator, defined as in PASCAL [19] by $m \ldots n = \{i : \mathbb{Z} \mid m \leq i \leq n\}$ for integer $m$ and $n$.

This (re)definition of $\sum_{i=m}^n e$ eliminates all possible misinterpretations. Moreover, since our notation does not require indexing dummies, it supports useful notational shorthands that hitherto required abuse of notation.

(4.17) Examples We illustrate how some typical shorthands that are considered “loose” even in traditional mathematics are made formally correct. Observe also that, in expressions of the form $\sum f$, the domain of the function $f$ appearing as the argument of $\sum$ need not be numeric.

- Let $ry$ and $ly$ be functions that map the (names of the) months in a regular year or a leap year to the number of days in that month. Then we can write expressions like $\sum ry = 365$ and $\sum ly = 366$. 
- If $i_K$ is the family of the currents flowing into node $K$ of an electrical network, Kirchhoff’s current law can be rigorously written $\sum i_K = 0$.
- The sum over any $\mathbb{C}$-valued data structure $s$ (list, graph, tree) is $\sum s$, since in our functional framework we define structures as functions.

Much more important than just a convenient and rigorous style of expression are the rules for formal calculation. Our operator supports both the point-wise and the point-free styles, as shown next.

The defining axioms for $\sum$ were (0) the empty rule, (1) the one-point rule, (2) the merge rule. Together with a direct consequence, namely (3) the composition rule, these are summarized in the following table in pointwise and point-free form.

<table>
<thead>
<tr>
<th></th>
<th>Point-free formulation</th>
<th>Point-wise formulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\sum e = 0$</td>
<td>$\sum (i:\emptyset.e) = 0$</td>
</tr>
<tr>
<td>1</td>
<td>$\sum (a \mapsto c) = c$</td>
<td>$\sum (i:i.j.e) = e[i := j]$</td>
</tr>
<tr>
<td>2</td>
<td>$\sum (f \cup g) = \sum f + \sum g$</td>
<td>$\sum (i: I \cup J.e) = \sum (i: I.e) + \sum (i: J.e)$</td>
</tr>
<tr>
<td>3</td>
<td>$\sum (f \circ g) = \sum f$</td>
<td>$\sum (i: I.e) = \sum (j : J.e[i := h,j])$</td>
</tr>
</tbody>
</table>

In all cases, we obviously assume suitable types for arguments and expressions.

For the merge rule, additional conditions are $\mathcal{D} f \cap \mathcal{D} g = \emptyset$ and $I \cap J = \emptyset$ respectively. The condition $\mathcal{D} f \cap \mathcal{D} g = \emptyset$ can be relaxed to $f \circ g$, in which case $\sum (f \cup g) = \sum f + \sum g - \sum (f \cap g)$, an evident analogy with a well-known rule in probability theory.

For the composition rule, additional conditions are that $\mathcal{D} f \subseteq \text{Bran} g$ and that $h \upharpoonright J$ is bijective on $I$. The pointwise version of this rule is the familiar change of dummy technique.

The most frequent application in everyday practice is the case where the argument function is a number sequence, for which the one-point and merge rules specialize to

$\sum (\tau \cdot c) = c \quad \text{and} \quad \sum (x \uplus y) = \sum x + \sum y.$

Both rules can be captured by the single rule

$\sum (x \prec c) = \sum x + c,$

and similarly for prefixing (exercise). As an example, instantiating (4.19) using

$(i: 0 \ldots n. f \ i) = (i: 0 \ldots n - 1. f \ i) \prec f \ n$

(for any function $f$ whose domain contains $0 \ldots n$) yields the pointwise equivalent

$\sum (i: 0 \ldots n. f \ i) = \sum (i: 0 \ldots n - 1. f \ i) + f \ n.$

This is called the domain split rule.

A very extensive overview of calculation rules, expressed in the point-wise style of the traditional notational conventions, can be found in (i) the book by Gries and Schneider [14] (ii) the book by Graham, Knuth and Patashnik [11].

We leave their derivation in our functional framework as an exercise.
The transposition operator  This example is interesting since it provides a
design example for a generic elastic operator, whereas the preceding ones were
domain-specific (\forall for booleans, \Sigma for numbers).

The image definition for the transposition operator \(-^T\) is \(f^T y x = f x y\),
where \(f\) is a “suitable” family of functions. The simplest suitable argument type
is \(A \to B \to C\) so that, given \(f : A \to B \to C\), then \(f^T \in B \to C \to A\). However,
this is too restrictive, since it does not even support dependent types for \(f\).

True to the general design principles for generic functionals from Section 2.1.2,
we want \(f^T\) to be defined for any family \(f\) of functions and, instead of imposing
restrictions on \(f\), we provide precise characterizations for \(D f^T\) and, given \(y : D f^T\),
for \(D (f^T y)\). We explore two design options and motivate the final choice.

(i) The most liberal choice for \(D f^T\) is clearly \(\bigcup x : D f . D (f x)\) or, equivalently,
\(\bigcup (D f)\). For any \(y\) in this domain, \(D (f^T y)\) is a subset of \(D f\) and, if
we want the equality \(f^T y x = f x y\), \(D (f^T y)\) to contain no out-of-domain applications,
\(D (f^T y) = \{x : D f \mid y \in D (f x)\}\). This leads to the first candidate definition

\[
def -^T : \text{fam } F \ni f \to \bigcup (D f) \ni y \to \{x : D f \mid y \in D (f x)\} \ni x \to (f x y)
\]
or, equivalently, \(-^T : \text{fam } F \to \text{fam } F\) together with the image definition (written
as an abstraction) \(f^T = y : \bigcup (D f) . x : \{x : D f \mid y \in D (f x)\} . f x y\).

Although this is the most general design choice and possibly useful in certain
situations, it is not the preferred one for our purposes.

(ii) The preferred choice is directed by one of the most important applications
of \(T\), namely in the generalization of the direct extension operator \(\simeq\) to certain
elastic operators \(F\). The requirement here is that properties of the form

\[
f \hat{\ast} g \hat{\ast} h = \hat{F}(f, g, h)
\]
holds for the variadic shorthand (for any number of arguments). In elastic operator
design, the general case can often be inferred from considering pairs. The type
of any pair of functions \(f, g\) can be written \(B \ni i \to (D f, D g) \ni i \to (R f, R g) i\). It is
a routine exercise to show that \(f \hat{\ast} g = (\ast) \circ (f, g)^T\) provided the type of \((f, g)^T\)
is \((D f \cap D g) \ni i \to (R f, R g) i\), where we draw attention to the fact that
intersection rather than union is required. This is because \((f \hat{\ast} g) x = f x \ast g x\)
contains no out-of-domain applications only if \(x \in D f \cap D g\).

Our preferred transposition operator imposes the requirements imposed by the
intended applications, hence the chosen definition is

\[
(4.21) \ def -^T : \text{fam } F \ni f \to \bigcap (D f) \to D f \ni x \to R (f x) \text{ with } f^T y x = f x y.
\]

Applications of this operator in various areas, from functional programming
languages to signal flow networks, will be exemplified in the courses using it.

Obvious designs All ad hoc abstractions from common notation are subsumed
by elastic operators and function abstraction. Hence, whenever we use expressions
similar to common notation, an elastic operator is tacitly assumed. Usually
the design is obvious, e.g., for any family \(S\) of sets, \(y \in \bigcap S \equiv \forall x : D S . y \in S x\) and
\(y \in \bigcup S \equiv \exists x : D S . y \in S x\). Filling in such details is left to the reader.
4.0.6 Overloading and polymorphism

Overloading has been present *avant la lettre* in traditional mathematical discourse for a long time, but has required more precise characterization due to developments in mathematics and programming.

The basic notion is simple: overloading a symbol (identifier) means using that symbol for designating “different” objects. Of course, for simple objects, this is possible only in different contexts, or in very informal contexts where the intended designation can be inferred, since otherwise ambiguity would result.

For operators\(^4\), the structure of the designated objects (functions) can be used for disambiguation, even in formal contexts. If the functions designated by the overloaded operator have different types but formally the same image definition, this form of overloading is called polymorphism. For instance, with the formal image definition \( \text{max} (x, y) = (x \leq y) \land y \neq x \), the operator \( \text{max} \) can be applied to pairs of various different types with possibly different total orderings \( \leq \). Hence considering general overloading also suffices for covering polymorphism.

Overloading involves two main issues: disambiguation, making the application of the overloaded operator to all its possible arguments unambiguous, and refined typing, the ability to reflect the type information of the designated functions in the operator’s type.

**Overloading by explicit parametrization** Sometimes a problem can be solved by circumventing it. Using a single operator to represent various functions can be done in a trivial way by adding an auxiliary parameter that directly or indirectly indicates the intended function. The operator for achieving this is already available, namely \( \times \). An example is the add parity bit function

\[
\text{def } \text{apb} : \times n : \mathbb{N} . \mathbb{B}^n \rightarrow \mathbb{B}^{n+1} \text{ with } \forall n : \mathbb{N} . \forall x : \mathbb{B}^n . \text{apb}_n x = x \leftarrow \bigoplus x
\]

Only the type definition is of importance here, the image definition just serves to make the example “concrete”. Observe that \( \text{apb}_n \in \mathbb{B}^n \rightarrow \mathbb{B}^{n+1} \) for any \( n \in \mathbb{N} \). Polymorphism with an auxiliary parameter is *Church-style polymorphism*.

**Overloading without auxiliary parameter** Polymorphism without auxiliary parameter is *Curry-style polymorphism*. We want an operator \( \otimes \) such that a parameterless variant \( \text{apb} \) can be written

\[
\text{def } \text{apb} : \otimes n : \mathbb{N} . \mathbb{B}^n \rightarrow \mathbb{B}^{n+1} \text{ with } \forall n : \mathbb{N} . \forall x : \mathbb{B}^n . \text{apb} x = x \leftarrow \bigoplus x
\]

This is achieved by the *function type merge* (\( \otimes \)) due to Van den Beuken \[29\].

\[
(4.22) \quad \text{def } \otimes : \text{fam} (\mathcal{P} \mathcal{F}) \rightarrow \mathcal{P} \mathcal{F} \text{ with } \otimes F = \{ \cup f \mid f : (\times F) \otimes \}.
\]

The filter \( \otimes \) is the elastic extension of \( \otimes \). It is defined as expected by \( \otimes f \equiv \forall (x, y) : (\mathcal{D} f)^2 . f x \otimes f y \) for any family \( f \) of functions. Definition (4.22) looks cryptic, but its principle is readily understood by elaborating \( \otimes (S, T) \) for given function types \( S \) and \( T \). Application examples appear in the two courses.

\(^4\)As usual, we use the term *operator* as a synonym for *function identifier*.
4.1 Introduction to concrete relation calculus

We introduce concepts about relations that are important in mathematics and computing. Our calculus is concrete in the sense that relations are not abstract entities but as a \( \mathbb{B} \)-valued functions, and operators over them are functionals. Yet, the point-free formulations of the various properties of these functionals are formally the same as the axioms of the most important abstract relational calculi.

4.1.0 Relations between sets: attributes and basic operators

We start with general relations between elements of possibly different sets, and specialize later to relations between elements of a single set.

(4.23) Definition: Relations A relation on a cross product \( \times S \) (where \( S \) is a finite list of sets) is a function of type \( \times S \to \mathbb{B} \). A relation of type \( C \times D \to \mathbb{B} \) is called a dyadic relation from \( D \) to \( C \).

(4.24) Examples of (dyadic) relations

a. The false relation on \( C \times D \) is the function \((C \times D) \cdot \mathbf{0}\)
b. The identity relation \( I_D \) on \( D \) is defined by \( I_D = (x, y) : D^2 \cdot (x = y) \)
c. Relation parent : \( P^2 \to \mathbb{B} \) where \( P \) is the set of persons.
d. The relation \( -\text{succ}- : \mathbb{Z}^2 \to \mathbb{B} \) with \( y \text{ succ } x \equiv y = x + 1 \)
e. The relation \( -\text{pred}- : \mathbb{Z}^2 \to \mathbb{B} \) with \( y \text{ pred } x \equiv y = x - 1 \)

(4.25) Convention Conjunctionality \( (b R c R d) \equiv b R c \land c R d \) is not imposed a priori in our formulation. Variadic shorthand for relational operators is always defined in terms of some elastic operator, for instance

\[
\begin{align*}
  a = b = c & \equiv \con (a, b, c) \\
  a \neq b \neq c & \equiv \text{inj} (a, b, c)
\end{align*}
\]

The conjunctionality of = and the “superconjunctionality” of \( \neq \) in the sense that \( a \neq b \neq c \equiv a \neq b \land b \neq c \land c \neq a \) follows from these definitions.

(4.26) Definition: Attributes of relations The attributes source (src), target (tgt), domain (dom) and range (ran) are defined as follows. For any dyadic relation \( R \),

\[
\begin{align*}
  \text{src } R & = (\{d \in R \mid \exists c : t R \cdot \text{src } c R d\}) \\
  \text{tgt } R & = (\{d \in R \mid \exists c : t R \cdot \text{tgt } c R d\}) \\
  \text{dom } R & = (\{d \cdot \text{src } R \mid \exists c : t R \cdot c R d\}) \\
  \text{ran } R & = (\{c \cdot \text{tgt } R \mid \exists d : \text{src } c R d\})
\end{align*}
\]

Defining a relation as a function includes information regarding the domain \((C \times D, \text{say})\). For instance, the empty relation on \( C \times D \) is \((C \times D) \cdot \mathbf{0}\) and satisfies \( \mathcal{D} ((C \times D) \cdot \mathbf{0}) = C \times D \). In formulations where such a relation is defined
as a subset of $C \times D$, this domain information must be handled separately since, for instance, the empty relation is then just $\mathcal{0}$.

A relation algebra isomorphic to relations as sets is obtained as follows.

(4.27) Definition: Pruned relations and set representations A relation $R$ is pruned iff $\mathcal{D}R = \text{ran} R \times \text{dom} R$. Letting $\text{Prel}$ be the type of pruned relations and $\text{Srel}$ the type of set representations of relations,

$$\text{def } \rho : \text{Prel} \rightarrow \text{Srel} \text { with } \rho R = \{(c, d) : \mathcal{D}R \mid c \mathcal{R} d\}$$

We do not normally assume that relations are pruned.

(4.28) Definitions: Reverse and Complement of relations

- The reverse operator $(-\sim)$ (“wok”). For any dyadic relation $R$,

$$\mathcal{D}R^{-} = \text{src} R \times \text{tgt} R \quad \text{and} \quad \forall (d, c) : \mathcal{D}R^{-}.dR^{-}c \equiv cRd$$

- The complement operator $(-\lnot)$ (“not”). For any dyadic relation $R$,

$$\mathcal{D}R^{\lnot} = \mathcal{D}R \quad \text{and} \quad \forall (c, d) : \mathcal{D}R^{\lnot}.cR^{\lnot}d \equiv \lnot(cRd)$$

Examples are $\text{succ}^{-} = \text{pred}$ and $(I_D)^{-} = I_D$. Here are some typical properties.

(4.29) Theorem: Some properties of the reversion operator

- $\text{dom} R^{-} = \text{ran} R$
- $\text{ran} R^{-} = \text{dom} R$
- $\mathcal{D}R^{-} = (\times \circ \text{rev} \circ \times^{-}) (\mathcal{D}R)$, provided $\mathcal{D}R$ is nonempty
- $(R^{-})^{-} = R$

Note: rev is defined by $\text{rev} x = i : \mathcal{D}x.x(#x - 1 - i)$ for any finite tuple $x$.

4.1.1 Relations on a set

A. Concrete relation algebra

We consider relations over a set $X$, and define the set $\text{rel}_X$ of all such relations by $\text{rel}_X = X^2 \rightarrow \mathcal{B}$. In principle, this does not reduce generality since $X$ can be taken to include all sets of interest. In applications, $X$ is usually some specific domain. All operators we define are implicitly polymorphic over $X$, i.e., type expressions of the form $T_X$ are taken as shorthand for $\otimes X : T \cdot T_X$ to avoid clutter.

(4.30) Definitions: Boolean and relational operators The operators $\sqcap$ and $\sqcup$ have type $(\text{rel}_X)^2 \rightarrow \text{rel}_X$, and $\boxdot$ has type $(\text{rel}_X)^2 \rightarrow \mathcal{B}$.

- $\sqcap$ (meet) is defined by $x(R \sqcap S)y \equiv xRy \land xSy$
- $\sqcup$ (join) is defined by $x(R \sqcup S)y \equiv xRy \lor xSy$
- $\boxdot$ (subrelation) is defined by $R \boxdot S \equiv \forall (x, y) : X^2.xRy \Rightarrow xSy$
It is easy to show that the first two operators, together with the complement operator, form a Boolean algebra. Also, $\subseteq$ is antisymmetric and, for relations $R$ and $S$ of the same type, $R \subseteq S \equiv R \cap S = R$ and $R \subseteq S \equiv R \cup S = S$.

(4.31) **Definitions: operators combining relations**  

- $\circ$ (composition) is defined by $x (R \circ S) z \equiv \exists y : X . x R y \land y S z$
- $\wedge$ (multiple composition) with $R^0 = I_X$ and $R^i(n + 1) = R \circ (R^i n)$.

If there is no danger for confusion, we abbreviate $R^\wedge n$ as $R^n$.

These definitions yield a rich relational algebra. Examples appear in the courses.

**B. Properties and Classification of Relations**

Relations may or may not have certain of the characteristics that are listed in Table 4.0, together with a predicate describing the considered characteristic. The columns labelled *predicate* and *image* contain completions for the similarly named placeholders in the following definition, recalling that $\text{Rel} = \bigcup X : T . \text{rel}_X$.

\[ \text{def predicate : Rel} \to \mathbb{B} \quad \text{with predicate} \ R \equiv \ \text{image} \quad \text{where} \quad X \ := \ \text{src} \ R \]

For instance, the first line of the table yields

\[ \text{def} \ \text{Refl} : \text{Rel} \to \mathbb{B} \quad \text{with} \quad \text{Refl} \ R \equiv \ \forall x : \text{src} \ R . \ x \ R \ x \]

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>predicate</th>
<th>image</th>
</tr>
</thead>
<tbody>
<tr>
<td>reflexive</td>
<td>Refl</td>
<td>$\forall x : X . \ x \ R \ x$</td>
</tr>
<tr>
<td>irreflexive</td>
<td>Irfl</td>
<td>$\forall x : X . \neg (x \ R \ x)$</td>
</tr>
<tr>
<td>symmetric</td>
<td>Symm</td>
<td>$\forall (x, y) : X^2 . \ x \ R \ y \Rightarrow y \ R \ x$</td>
</tr>
<tr>
<td>asymmetric</td>
<td>Asym</td>
<td>$\forall (x, y) : X^2 . \ x \ R \ y \Rightarrow \neg (y \ R \ x)$</td>
</tr>
<tr>
<td>antisymmetric</td>
<td>Ants</td>
<td>$\forall (x, y) : X^2 . \ x \ R \ y \Rightarrow y \ R \ x \Rightarrow x = y$</td>
</tr>
<tr>
<td>transitive</td>
<td>Trns</td>
<td>$\forall (x, y, z) : X^3 . \ x \ R \ y \Rightarrow y \ R \ z \Rightarrow x \ R \ z$</td>
</tr>
<tr>
<td>total</td>
<td>Totl</td>
<td>$\forall (x, y) : X^2 . \ x \ R \ y \lor y \ R \ x$</td>
</tr>
<tr>
<td>total disorder</td>
<td>Dsrd</td>
<td>$\forall (x, y) : X^2 . \ x \ R \ y \Rightarrow x = y$</td>
</tr>
<tr>
<td>equivalence</td>
<td>EQ</td>
<td>$\text{Trns} \ R \land \text{Ref} \ R \land \text{Symm} \ R$</td>
</tr>
<tr>
<td>preorder</td>
<td>PR</td>
<td>$\text{Trns} \ R \land \text{Ref} \ R$</td>
</tr>
<tr>
<td>partial order</td>
<td>PO</td>
<td>$\text{PR} \ R \land \text{Ants} \ R$</td>
</tr>
<tr>
<td>total order</td>
<td>TO</td>
<td>$\text{PO} \ R \land \text{Totl} \ R$</td>
</tr>
<tr>
<td>quasi order</td>
<td>QO</td>
<td>$\text{Trns} \ R \land \text{Irfl} \ R$</td>
</tr>
<tr>
<td>strict p.o.</td>
<td>—</td>
<td>(Synonym of quasi order)</td>
</tr>
<tr>
<td>well-founded</td>
<td>WF</td>
<td>see Definition 4.58 on page 87</td>
</tr>
<tr>
<td>well-order</td>
<td>WO</td>
<td>$\text{WF} \ R \land \text{Trns} \ R$</td>
</tr>
</tbody>
</table>

Table 4.0: Classification predicates over relations

**C. Extensions and Closures**

It is advantageous to introduce a few auxiliary operators that facilitate definitions and calculations by handling the properties of interest (reflexivity, transitivity and so on) as parameters.
4.1. Introduction to concrete relation calculus

For any predicate \( P : \text{rel}_X \to \mathbb{B} \) on relations over \( X \), we define the relations \( \mathcal{E}_P \) and \( \mathcal{C}_P \) of type \( \text{rel}_X^2 \to \mathbb{B} \) as follows.

- \( R' \) is a \( P \)-extension of \( R \) iff \( R' \mathcal{E}_P R \) where

\[
(4.32) \quad R' \mathcal{E}_P R \equiv P R' \land R \sqsubseteq R'
\]

- \( R' \) is a \( P \)-closure of \( R \) iff \( R' \mathcal{C}_P R \) where

\[
(4.33) \quad R' \mathcal{C}_P R \equiv R' \mathcal{E}_P R \land \forall R'' : \text{rel}_X . R'' \mathcal{E}_P R \Rightarrow R' \sqsubseteq R''
\]

So \( R' \mathcal{C}_P R \) iff \( R' \) is the least \( P \)-extension of \( R \). It is easy to show that \( \sqsubseteq \) is antisymmetric, and that therefore closures (least elements under \( \sqsubseteq \)) are unique.

\[ (4.34) \text{Theorem: closure uniqueness \quad } R' \mathcal{C}_P R \land R'' \mathcal{C}_P R \Rightarrow R' = R'' \]

This makes the functions introduced in the following definitions well-defined

\[ (4.35) \text{Definition: closure operators \quad } \text{We define the following operators, all of type } \text{rel}_X \to \text{rel}_X:\]

- \( \refl \), the reflexive-closure operator, with \( R^\refl = \mathcal{C}_\refl R \)
- \( \symm \), the symmetric-closure operator, with \( R^\symm = \mathcal{C}_\symm R \)
- \( \trans \), the transitive-closure operator, with \( R^\trans = \mathcal{C}_\trans R \)
- \( \ast \), the reflexive and transitive-closure operator, with \( R^\ast = (R^\refl)^\trans \)

It is an instructive exercise to prove that \( (R^\refl)^\trans = (R^\trans)^\refl \) but not necessarily \( (R^\symm)^\trans = (R^\trans)^\symm \), and to investigate the relationship between \( (R^\symm)^\refl \) and \( (R^\refl)^\symm \).

It is also easy to prove that \( \refl \Rightarrow R^\refl = R \) and \( \symm \Rightarrow R^\symm = R \) and \( \trans \Rightarrow R^\trans = R \). Explicit characterizations are given next.

\[ (4.36) \text{Theorem: properties of the closure operators}\]

\[
\begin{align*}
R^\refl &= R^0 \cup R^1 = \text{I}_X \uplus R \\
R^{\symm} &= R \cup R^\trans \\
x R^+ y &\equiv \exists n : \mathbb{N}_{>0} . x R^n y \\
R^\ast &= R^0 \uplus R^+ = \text{I}_X \uplus R^+
\end{align*}
\]

4.1.2 Order relations

For a good understanding of order relations, it is advantageous not to start with relations that already satisfy all properties of a partial order (reflexivity, transitivity, antisymmetry), but rather to introduce properties gradually, to the effect that the ramifications of each property can be appreciated separately.

Arbitrary general relations will be denoted by \( \preceq \) or \( \sqsubseteq \) (we may want different symbols in case different relations appear in the same context), and arbitrary reflexive relations will be similarly denoted by \( \preceq \) or \( \sqsubseteq \).
A. TWO STYLES OF FORMULATION: SETS VERSUS PREDICATES

As a preamble, we observe that, in the literature, many concepts of interest to the current topic are formulated in terms of sets. Since the calculation rules for sets are usually (and not only in our formalism) expressed via the calculation rules for proposition and predicate calculus, one of the first steps in formal calculation is switching from sets to predicates, and one of the last steps is switching back to sets (the latter only in case an expression with sets is required).

Formulations with sets are equivalent to formulations with predicates, as shown by the existence of operators \( \text{set} \) and \( \text{prd} \) defined such that

\[
(4.37) \quad \text{def set}_\exists : \mathcal{T} \ni X \rightarrow (X \rightarrow \mathbb{B}) \rightarrow \mathcal{P} \, X \text{ with } \text{set}_X \, P = \{ x : X \mid P \, x \}
\]

\[
(4.38) \quad \text{def prd}_\exists : \mathcal{T} \ni X \rightarrow \mathcal{P} \, X \rightarrow (X \rightarrow \mathbb{B}) \text{ with } \text{prd}_X \, S \, x = x \in S.
\]

In other words, for any set \( X \), \( \text{set}_X \in (X \rightarrow \mathbb{B}) \rightarrow \mathcal{P} \, X \), transforming predicates over \( X \) into subsets of \( X \), and \( \text{prd}_X \in \mathcal{P} \, X \rightarrow (X \rightarrow \mathbb{B}) \), the inverse (exercise).

To avoid the aforementioned switching between sets and predicates, which is simple but conceptually irrelevant and practically wasteful, we shall mostly use formulations with predicates, except in the beginning to illustrate both styles.

We reduce writing by introducing operators \( \text{pred} \) and \( \text{rel} \): for any set \( X \),

\[
(4.39) \quad \text{pred}_X = X \rightarrow B \quad \text{and} \quad \text{rel}_X = X^2 \rightarrow B.
\]

The set of all predicates on a set and the set of all relations on a set are then

\[
(4.40) \quad \text{Pred} := \bigcup \, X : \mathcal{T} \cdot \text{pred}_X \quad \text{and} \quad \text{Rel} := \bigcup \, X : \mathcal{T} \cdot \text{rel}_X.
\]

Observe that any \( R : \text{Rel} \) satisfies \( \text{tgt} \, R = \text{src} \, R \).

B. EXTREMAL ELEMENTS

We present the definitions first in the set-oriented style.

\[
(4.41) \quad \text{Definitions: “low” extremal elements of a relation} \quad \text{Let} \quad \preceq \,
\]

\( \preceq \) : \text{Rel} and \( X := \text{src} \, (\prec) \). For any subset \( S \) of \( X \) and any \( x \) in \( X \),

- \( x \) is minimal in \( S \) iff \( x \) belongs to \( S \) and every \( y \) in \( X \) satisfying \( y \prec x \) does not belong to \( S \). Formally

\[
(4.41.a) \quad x \text{ ismin}_\preceq \, S \quad \equiv \quad x \in S \land \forall \, y : X \cdot y \prec x \Rightarrow y \notin S
\]

- \( x \) is a lower bound for \( S \) iff all elements \( y \) of \( S \) satisfy \( x \prec y \). Formally:

\[
(4.41.b) \quad x \text{ islb}_\preceq \, S \quad \equiv \quad \forall \, y : S \cdot x \prec y
\]

- \( x \) is a least element of \( S \) iff \( x \) is in \( S \) and is a lower bound for \( S \):

\[
(4.41.c) \quad x \text{ ileast}_\preceq \, S \quad \equiv \quad x \in S \land x \text{ islb}_\preceq \, S
\]

The type of these operators is (exercise) \( \text{Rel} \ni R \rightarrow \text{src} \, R \times \mathcal{P} \, (\text{src} \, R) \rightarrow \mathbb{B} \).
4.1. Introduction to concrete relation calculus

The third definition can also be written $x \text{ismin}_{\prec} S \equiv x \in S \land \forall y : S \cdot (y \prec x)$.

More elegant than the set formulation is the following one, based on predicate transformers of type $\text{pred}_X \rightarrow \text{pred}_X$.

(4.42) Definitions: “low” extremal elements of a relation Let $\prec : \text{Rel}$ and $X := \text{src}(\prec)$. For any predicate $P$ over $X$ and any $x$ in $X$, we say that

- $x$ is minimal for $P$ iff $x$ satisfies $P$ and every $y : X$ satisfying $y \prec x$ does not satisfy $P$. Formally

$$\text{min}_{\prec} P x \equiv P x \land \forall y : X \cdot y \prec x \Rightarrow \neg(P y)$$

- $x$ is a lower bound for $P$ iff $x \prec y$ for all elements $y : X$ satisfying $P$:

$$\text{lb}_{\prec} P x \equiv \forall y : X . P y \Rightarrow x \prec y$$

- $x$ is a least element for $P$ iff $x$ satisfies $P$ and is a lower bound for $P$:

$$\text{lst}_{\prec} P x \equiv P x \land \text{lb}_P x$$

Here the operators $\text{min}$, $\text{lb}$ and $\text{lst}$ all have type $\text{Rel} \ni R \rightarrow \text{pred}_{\text{src}R} \rightarrow \text{pred}_{\text{src}R}$.

Caution: words such as minimal and least, which are often used interchangeably in natural language, are given a precise and distinct meaning in mathematics. Hence, any connotations other than those implied by the formal definitions should be carefully avoided. In particular, don’t be misled by a priori associations with familiar ordering relations, such as those for numbers. Instructive examples in this respect are: (i) For a reflexive relation, a set or relation cannot have minimal elements, (ii) No element can be both least and minimal.

The following definitions swap the arguments of the relational operator.

(4.43) Definition: “high” extremal elements of a relation We define upper bound, greatest element and maximal element as follows.

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator type: $rel_X : X \times \mathcal{P} X \rightarrow \mathbb{B}$</td>
<td>$\text{isub}_{\prec}$ $S$</td>
<td>$\forall y : S \cdot y \prec x$</td>
</tr>
<tr>
<td></td>
<td>$\text{isgrst}_{\prec}$ $S$</td>
<td>$x \in S \land \text{isub}_{\prec} S$</td>
</tr>
<tr>
<td></td>
<td>$\text{ismax}_{\prec} S$</td>
<td>$x \in S \land \forall y : X . x \prec y \Rightarrow y \notin S$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator type: $rel_X : X \rightarrow \text{pred}_X$</td>
<td>$\text{ub}_{\prec}$ $P x$</td>
<td>$\forall y : X . P y \Rightarrow y \prec x$</td>
</tr>
<tr>
<td></td>
<td>$\text{gst}_{\prec} P x$</td>
<td>$P x \land \text{ub}_P x$</td>
</tr>
<tr>
<td></td>
<td>$\text{max}_{\prec} P x$</td>
<td>$P x \land \forall y : X . x \prec y \Rightarrow \neg(P y)$</td>
</tr>
</tbody>
</table>

C. Reflexive relations

By definition, a relation $\leq : X^2 \rightarrow \mathbb{B}$ is reflexive iff $\forall x : X . x \leq x$. An alternative characterization is given by the following theorem (proof: exercise).
(4.44) **Theorem**  \( \text{Refl}(\preceq) \equiv \forall (x, y) : X^2 . x = y \Rightarrow x \preceq y \)

**Corollary**  \( \text{Refl}(\preceq) \equiv \forall (x, y) : X^2 . x = y \Rightarrow x \preceq y \land y \preceq x \)

(4.45) **Theorem:** Laws of indirect order (EWD1240)  A reflexive relation \( \preceq : \text{rel}_X \) satisfies

\[
\forall (z : X . z \preceq x \Rightarrow z \preceq y) \Rightarrow x \preceq y \\
\forall (z : X . y \preceq z \Rightarrow x \preceq z) \Rightarrow x \preceq y 
\]

For any reflexive \( \preceq : \text{rel}_X \) we can define an irreflexive relation \( \prec : \text{rel}_X \) by \( x \prec y \equiv x \preceq y \land \lnot (y \preceq x) \), which has the property that \( x \text{isleast}_{\prec} S \Rightarrow x \text{ismin}_{\prec} S \).

**D. Antisymmetric relations**

By definition, \( \prec : \text{rel}_X \) is antisymmetric iff \( \forall (x, y) : X^2 . x \prec y \land y \prec x \Rightarrow x = y \).

We have already seen that antisymmetry entails uniqueness of least elements.

(4.46) **Theorem:** Uniqueness of least elements  If \( \prec : \text{rel}_X \) is antisymmetric, then, for any subset \( S \) of \( X \),

\[ x \text{isleast}_{\prec} S \land y \text{isleast}_{\prec} S \Rightarrow x = y \]

**E. Relations that are both reflexive and antisymmetric**

It is easy to prove that reflexivity and antisymmetry can be combined in the single characterization

(4.47) **Theorem:** Combining reflexivity and antisymmetry

\[ \text{Refl}(\preceq) \land \text{Ants}(\preceq) \equiv \forall (x, y) : X^2 : x \preceq y \land y \preceq x \equiv x = y \]

An important property is the following.

(4.48) **Theorem:** Laws of indirect equality (EWD1240)  A relation \( \preceq : \text{rel}_X \) that is reflexive and antisymmetric satisfies

\[
\forall (z : X . z \preceq x \equiv z \preceq y) \Rightarrow x = y \\
\forall (z : U . x \preceq z \equiv y \preceq z) \Rightarrow x = y
\]

**F. Least upper bounds and greatest lower bounds**

Least upper bounds (and their dual, greatest lower bounds) play a central role in many areas of mathematics and in practically all areas of computing science. The definitions are formal transliterations of the names.

(4.49) **Definition:** Least upper bounds, greatest lower bounds

(again expressed in the two formulations)
4.1. Introduction to concrete relation calculus

<table>
<thead>
<tr>
<th>operator type: ( \text{rel}_X \to X \times \mathcal{P} X \to \mathbb{B} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \text{ islub}_\prec S \equiv \forall y : X . x \preceq y \equiv \forall s : S . s \preceq y )</td>
</tr>
<tr>
<td>( x \text{ isglb}_\prec S \equiv \forall y : X . x \succeq y \equiv \forall s : S . s \succeq y )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operator type: ( \text{rel}_X \to \text{pred}_X \to \text{pred}_X )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{lub}<em>\prec P \equiv \text{lst}</em>\prec (\text{ub}_\prec P) )</td>
</tr>
<tr>
<td>( \text{glb}<em>\prec P \equiv \text{gst}</em>\prec (\text{lb}_\prec P) )</td>
</tr>
</tbody>
</table>

We have shown earlier that antisymmetry is a sufficient condition for uniqueness. If the uniqueness condition is satisfied, then we define the least upper bound operator \( (\bigcup_\prec) \) and greatest lower bound operator \( (\bigcap_\prec) \) as follows

(4.50) **Definition:** The LUB \( (\bigcup_\prec) \) and GLB \( (\bigcap_\prec) \) Operators For any family \( F \) of elements \( X \) such that the l.u.b. or g.l.b. exists for \( R F \),

\[
\bigcup_\prec F \text{ islub}_\prec R F \quad \bigcap_\prec F \text{ isglb}_\prec R F
\]

Since these operators are elastic, their variadic counterparts are also evident, viz, \( x \bigcup y \bigcup z = \bigcup (x, y, z) \). Here we used the following convention.

(4.51) **Convention** Often the argument \( \prec \) in \( \bigcup_\prec \) or \( \bigcap_\prec \) is understood and therefore omitted in the literature. We shall follow this practice.

The order in which the definitions are particularized to relations with special properties (reflexivity etc.) is a matter of taste or opportunity.

(4.52) **Examples**

- If relation \( \preceq \) is reflexive and transitive, it is easy to show that
  
  \[
  x \text{ islub}_\prec S \equiv \forall y : X . x \preceq y \equiv \forall s : S . s \preceq y
  \]
  
  \[
  \text{lub}_\prec P x \equiv \forall y : X . x \preceq y \equiv \forall s : X . P s \Rightarrow s \preceq y
  \]
  
  In fact, in the proof the conditions of reflexivity and transitivity arise naturally, and are easily discovered even if they are not stated a priori.

- If a relation \( \sqsubseteq \) is reflexive and antisymmetric, and we define the relation \( \text{ilu}_\sqsubseteq \) between elements of \( X \) and pairs of elements of \( X \) by
  
  \[
  \text{def} \quad \text{ilu}_\sqsubseteq : X \times X^2 \to \mathbb{B}
  \]
  
  \[
  x \text{ ilu}_\sqsubseteq (u, v) \equiv \forall z : X . x \sqsubseteq z \equiv u \sqsubseteq z \land v \sqsubseteq z
  \]
  
  then it is easy to prove uniqueness of the l.u.b. in the sense that
  
  \[
  x \text{ ilu}_\sqsubseteq (u, v) \land y \text{ ilu}_\sqsubseteq (u, v) \Rightarrow x = y
  \]
  
  This justifies defining the l.u.b. operator \( \bigcup \) by
  
  \[
  (u \bigcup v) \text{ ilu}_\sqsubseteq (u, v)
  \]
  
  or, equivalently (eliminating \( \text{ilu}_\sqsubseteq \))
  
  \[
  \forall z : X . (u \bigcup v \sqsubseteq z \equiv u \sqsubseteq z \land v \sqsubseteq z)
  \]
4.2 Induction principles

Mathematical induction is one of the most powerful proof techniques in pure and applied mathematics. Complete automatic theorem-provers (Boyer Moore, Nuprl etc.) are based on this technique.

We start by recalling the induction principle over natural numbers in the form that should be familiar from high school and other mathematics courses. The main purpose of this refresher is pointing out certain pitfalls that are perpetuated by the customary informal approach, and establishing a systematic formal discipline that helps avoiding errors.

Subsequently, we present a more general theory of induction, linking it to well-founded relations, and deriving all inductive methods of interest as special cases. Emphasis will be on non-numeric and structural induction.

4.2.0 Induction over the natural numbers

A. Weak induction

We recall the following induction principle from high school. The fact that this principle is sound for an arbitrary predicate \( P : \mathbb{N} \rightarrow \mathbb{B} \) will be proven later.

\[
(4.53) \text{Definition: Weak Induction Principle (over } \mathbb{N}\text{)}
\]

\[ P \; 0 \land \forall (n : \mathbb{N} . P n \Rightarrow P (n + 1)) \equiv \forall (n : \mathbb{N} . P n) \]

So, in principle, proving \( \forall n : \mathbb{N} . P n \) is equivalent to proving the following.

- (BC) Base case: prove \( P 0 \)
- (IC) Inductive case: prove \( \forall (n : \mathbb{N} . P n \Rightarrow P (n + 1)) \).

In practice, the proof of the inductive case is further (and often tacitly) reduced in two stages.

a. To prove \( \forall (n : \mathbb{N} . P n \Rightarrow P (n + 1)) \), one proves \( n \in \mathbb{N} \Rightarrow P n \Rightarrow P (n + 1) \), implicitly followed by generalization.

b. To prove \( n \in \mathbb{N} \Rightarrow P n \Rightarrow P (n + 1) \) one uses the deduction theorem (1.11): assuming the antecedents \( n \in \mathbb{N} \) and \( P n \) (the induction hypothesis (IH)) one proves the consequent \( P (n + 1) \) in a so-called induction step (IS).

This is illustrated in the following classical “textbook example”.

\[
(4.54) \text{Example: proving } \forall n : \mathbb{N} . \sum (i : \Box n \cdot 2 \cdot i + 1) = n^2
\]

To this effect, we introduce \( P : \mathbb{N} \rightarrow \mathbb{B} \) with \( P n \equiv \sum (i : \Box n \cdot 2 \cdot i + 1) = n^2 \) and prove \( \forall n : \mathbb{N} . P n \) by induction.

- (BC) Proof of \( P 0 \), i.e., \( \sum (i : \Box 0 \cdot 2 \cdot i + 1) = 0^2 \)

\[
\sum (i : \Box 0 \cdot 2 \cdot i + 1) = \langle \text{Empty rule} \rangle \quad 0
\]

\[
= \langle \text{Arithmetic} \rangle \quad 0^2
\]
4.2. Induction principles

- (IC) will be proved in two ways. The first one involves only the aforementioned reduction a., so we prove $Pn \Rightarrow P(n+1)$ for $n : \mathbb{N}$.

\[ Pn \equiv \langle \text{Def. } P \rangle \sum (i : \square n . 2 \cdot i + 1) = n^2 \]
\[ \Rightarrow \langle \text{Leibniz} \rangle \sum (i : \square n . 2 \cdot i + 1) + 2 \cdot n + 1 = n^2 + 2 \cdot n + 1 \]
\[ \equiv \langle \text{Domain split} \rangle \sum (i : \square (n+1) . 2 \cdot i + 1) = n^2 + 2 \cdot n + 1 \]
\[ \equiv \langle \text{Arithmetic} \rangle \sum (i : \square (n+1) . 2 \cdot i + 1) = (n+1)^2 \]
\[ \equiv \langle \text{Def. } P \rangle \quad P(n+1) \]

Domain split means $\sum (i : 0..n . f i) = \sum (i : 0..n - 1 . f i) + f n$ for any numeric function $f$ such that $0..n \subseteq D f$ (see 4.20).

The second proof combines both reductions a. and b.

(IH) Assume $Pn$, i.e., $\sum (i : \square n . 2 \cdot i + 1) = n^2$

(IS) Proof of $P(n+1)$, i.e., $\sum (i : \square (n+1) . 2 \cdot i + 1) = (n+1)^2$

\[ \sum (i : \square (n+1) . 2 \cdot i + 1) \]
\[ = \langle \text{Domain split} \rangle \sum (i : \square n . 2 \cdot i + 1) + 2 \cdot n + 1 \]
\[ = \langle \text{IH} \rangle \quad n^2 + 2 \cdot n + 1 \]
\[ = \langle \text{Arithmetic} \rangle \quad (n+1)^2 \]

Note that the IH is $Pn$, certainly not $Pn \equiv \sum (i : \square n . 2 \cdot i + 1) = n^2$, which trivially holds by virtue of the definition of $P$.

**Warning** Applying the deduction theorem with $Pn$ as the hypothesis implies that, in the induction step, $n$ must be treated as a constant in the sense that it may not be instantiated. Note that, otherwise, the induction step would be trivial since instantiating $n$ with $n + 1$ would suffice. This warning may appear superfluous, yet it concerns one of the most frequent mistakes students make, often hidden within calculations that make it less conspicuous. The fact that the aforementioned reduction of the inductive case (using the deduction theorem and generalization) is nearly always done tacitly obscures what exactly is going on. Fortunately, the predicate calculus elucidates the situation.

The concern about instantiation is quite pertinent, since instantiation is precisely what often becomes necessary in case several quantified variables are involved. The power of the induction principle is such that it need be applied to only one variable, but the cost is that some extra care is necessary.

Indeed, assume one has to prove $\forall (m, n) : \mathbb{N}^2 . Q(m, n)$ for some predicate $Q : \mathbb{N}^2 \rightarrow \mathbb{B}$. By the rules of the predicate calculus (which ones?) this is equivalent to proving $\forall n : \mathbb{N} . Pn$ where $P : \mathbb{N} \rightarrow \mathbb{B}$ with $Pn \equiv \forall m : \mathbb{N} . Q(m, n)$. The induction hypothesis $Pn$ is obviously $\forall m : \mathbb{N} . Q(m, n)$, which means that, in the proof of the induction step, the $m$ of the induction hypothesis may be instantiated by any $\mathbb{N}$-valued expression but, as in the general case, $n$ must be treated as a constant because of the deduction theorem. In the induction step, the proof of $P(n+1)$, i.e., $\forall m : \mathbb{N} . Q(m, n+1)$, usually also involves implicit generalization in the sense that one actually proves $m \in \mathbb{N} \Rightarrow Q(m, n+1)$. This $m$ may obviously not be instantiated.
Again, the danger resides in the fact that all these proof-technical decisions are made tacitly, in order to reduce writing — certainly a worthwhile goal.

However, for this very reason, it becomes crucial (and also is a courtesy to the reader of the proof) to explicitly state the predicate $P$ and adhere to the proof conventions; therefore we shall insist on this discipline in all test and exams.

All the aforementioned subtleties are illustrated in the following example.

(4.55) Example: a property of the Fibonacci numbers  

Given

$$\text{def } F_\_ : \mathbb{N} \to \mathbb{N} \text{ with } F_0 = 0 \land F_1 = 1 \land F_{n+2} = F_{n+1} + F_n,$$

we want to prove that, for any $m$ and $n$ in $\mathbb{N}$,

$$F_{m+n+1} = F_{m+1} \cdot F_{n+1} + F_m \cdot F_n.$$  

We define $P : \mathbb{N} \to \mathbb{B}$ with $Pn \equiv \forall m : \mathbb{N}. F_{m+n+1} = F_{m+1} \cdot F_{n+1} + F_m \cdot F_n$ and prove $\forall n : \mathbb{N}. Pn$ by induction.

- (BC) Proof of $P0$, i.e., $\forall m : \mathbb{N}. F_{m+1} = F_{m+1} \cdot 1 + F_m \cdot 0$. This is a simple exercise since $F_1 = 1$ and $F_0 = 0$.

- (IC) Proof of $Pn \Rightarrow P(n + 1)$ for arbitrary $n : \mathbb{N}$.

(IH) Assume $Pn$, i.e., $\forall m : \mathbb{N}. F_{m+n+1} = F_{m+1} \cdot F_{n+1} + F_m \cdot F_n$.

(IS) $P(n + 1) \equiv \forall m : \mathbb{N}. F_{m+(n+1)+1} = F_{m+1} \cdot F_{(n+1)+1} + F_m \cdot F_{n+1}$, so we prove $F_{m+(n+1)+1} = F_{m+1} \cdot F_{(n+1)+1} + F_m \cdot F_{n+1}$ for $m : \mathbb{N}$.

- (Assoc. +) $F_{(m+1)+n+1}$

- (Instant. IH) $F_{m+2} \cdot F_{n+1} + F_{m+1} \cdot F_n$

- (Def. F) $(F_{m+1} + F_m) \cdot F_{n+1} + F_m \cdot F_n$

- (Arithmetic) $F_{m+1} \cdot F_{n+1} + F_m \cdot F_{n+1}$

- (Def. F) $F_{m+1} \cdot F_{n+2} + F_m \cdot F_{n+1}$

B. STRONG INDUCTION

A variant that is more powerful in practice (because it has a stronger antecedent inside the leftmost quantified expression) is called strong induction:

(4.56) Definition: Strong Induction Principle  (over $\mathbb{N}$)

$$\forall (n : \mathbb{N}. \forall (i : \square n. P i) \Rightarrow P n) \equiv \forall (n : \mathbb{N}. P n)$$

This form does not involve a separate base case since $P0$ is already implied by the left-hand side as can be seen from

$$\forall (n : \mathbb{N}. \forall (i : \square n. P i) \Rightarrow P n) \Rightarrow \langle \text{Instantiation } n := 0 \rangle \ \forall (i : \square 0. P i) \Rightarrow P 0$$

$$\equiv \langle \text{Empty rule} \rangle \ 1 \Rightarrow P 0$$

$$\equiv \langle \text{Left identity } \Rightarrow \rangle \ P 0,$$
which allows rewriting the basic form (4.56) as follows (exercise).

\[(4.57) \quad P \ 0 \land \forall (n : \mathbb{N}. \forall (i : 0 \ldots n. P i) \Rightarrow P (n + 1)) \quad \equiv \quad \forall (n : \mathbb{N}. P n)\]

Form (4.57) is useful when a base case is appropriate for some reason. However, often it is more elegant to avoid a separate base case, and directly use (4.56).

### 4.2.1 Well-founded relations and induction

**A. Principle**

A first definition states when a relation is well-founded.

\[(4.58) \text{Definition: Well-foundedness} \quad \text{A relation } \prec : X^2 \to \mathbb{B} \text{ is well-founded if every nonempty subset of } X \text{ has a minimal element.} \]

\[\text{WF} (\prec) \quad \equiv \quad \forall S : \mathcal{P} X . S \neq \emptyset \Rightarrow \exists x : X . x \text{ ismin}_\prec S\]

On the other hand, a second definition states what it means to support induction.

\[(4.59) \text{Definition: Relations supporting induction} \quad \text{A relation } \prec : X^2 \to \mathbb{B} \text{ supports induction iff SI } (\prec), \text{ which is defined by} \]

\[\text{SI} (\prec) \quad \equiv \quad \forall P : \text{pred}_X . \forall (x : X . \forall (y : X \downarrow x . P y) \Rightarrow P x) \Rightarrow \forall x : X . P x\]

Equivalence between the two is established by the following theorem.

\[(4.60) \text{Theorem} \quad \text{WF} (\prec) \equiv \text{SI} (\prec)\]

**Proof**

\[\text{WF} (\prec)\]

\[\equiv \quad (\text{Definition WF} (4.58) \text{ and } S \neq \emptyset \equiv \exists x : S . 1)\]

\[\forall S : \mathcal{P} X . \exists (x : S . 1) \Rightarrow \exists (x : X . x \text{ ismin}_\prec S)\]

\[\equiv \quad \langle S = X \cap S, \text{ trading} \rangle\]

\[\forall S : \mathcal{P} X . \exists (x : X . x \in S) \Rightarrow \exists (x : X . x \text{ ismin}_\prec S)\]

\[\equiv \quad (\text{Definition ismin} (4.41))\]

\[\forall S : \mathcal{P} X . \exists (x : X . x \in S) \Rightarrow \exists (x : X . x \in S \land \forall y : X . y < x \Rightarrow y \notin S)\]

\[\equiv \quad \langle p \Rightarrow q \equiv \neg q \Rightarrow \neg p \rangle\]

\[\forall S : \mathcal{P} X . \neg (\exists x : X . x \in S \land \forall y : X . y < x \Rightarrow y \notin S) \Rightarrow \neg (\exists x : X . x \in S)\]

\[\equiv \quad (\text{Duality } \forall / \exists, \text{ De Morgan})\]

\[\forall S : \mathcal{P} X . \forall (x : X . x \notin S \lor \neg (\forall y : X . y < x \Rightarrow y \notin S)) \Rightarrow \forall x : X . x \notin S\]

\[\equiv \quad \langle \lor \text{ to } \Rightarrow, \text{ i.e., } a \lor \neg b \equiv b \Rightarrow a \rangle\]

\[\forall S : \mathcal{P} X . \forall (x : X . \forall (y : X . y < x \Rightarrow y \notin S) \Rightarrow x \notin S) \Rightarrow \forall x : X . x \notin S\]

\[\equiv \quad (\text{Change of variables: } S = \{x : X \mid \neg (P x)\} \text{ and } P x \equiv x \notin S, \text{ exercise})\]

\[\forall P : X \to B . \forall (x : X . \forall (y : X . y < x \Rightarrow P y) \Rightarrow P x) \Rightarrow \forall x : X . P x\]

\[\equiv \quad (\text{Trading, def. SI} (4.59))\]

\[\text{SI} (\prec)\]
Yet another equivalent characterization for well-foundedness or supporting induction can be given via decreasing chains. Here we provide only an outline; the details are left as exercises. A sequence \( f: X^\omega \) is a decreasing chain w.r.t. a relation \( \prec: X^2 \to \mathbb{B} \), written \( \text{decr} f \), iff \( \forall (i, j): (\mathcal{D} f)^2. i > j \Rightarrow f_i \prec f_j \). The relation \( \prec \) is called Noetherian iff every such decreasing chain is finite (formalization left as an exercise). Theorem: a relation is well-founded iff it is Noetherian.

**B. PARTICULAR INSTANCES OF WELL-FOUNDED INDUCTION**

**Induction over \( \mathbb{N} \)** Most illustrative as a first example are the strong and weak induction principles for natural numbers presented earlier. We are now in a position to prove them axiomatically. One of the axioms for natural numbers is:

Every nonempty subset of \( \mathbb{N} \) has a least element under \( \leq \).

Since \( x \leq y \equiv \neg (y < x) \), it is easy to show that this axiom is equivalent to stating that every nonempty subset of \( \mathbb{N} \) has a minimal element under \( < \) (exercise).

Strong induction over \( \mathbb{N} \) immediately follows by instantiating Theorem 4.60 with \( < \) for \( \prec \), yielding the corollary

\[
\forall (n: \mathbb{N}. \ P n) \ \equiv \ \forall (n: \mathbb{N}. \forall (m: \mathbb{N}. \ m < n \Rightarrow P m) \Rightarrow P n)
\]

Weak induction over \( \mathbb{N} \) can be obtained in two ways. The first consists in proving that the relation \( \prec \) defined by

\[
m \prec n \equiv m + 1 = n
\]

is well-founded, and deducing from the general form in Definition 4.59 that

\[
\forall (n: \mathbb{N}. \ P n) \ \equiv \ P 0 \land \forall (n: \mathbb{N}. \ P n \Rightarrow P (n + 1)).
\]

The other approach consists in directly showing both principles to be equivalent.

**Structural induction** Over sequences, it is possible to define suitable prefix relations that are well-founded and hence support induction. Elaboration is left as an exercise. One of the most important resulting induction principles can be formulated as follows.

\[
(4.61) \ \text{Theorem: structural induction for lists} \quad \text{For any set} \ A \ \text{and any} \ P: A^* \to \mathbb{B},
\]

\[
\forall (x: A^*. \ P x) \ \equiv \ P \varepsilon \land \forall (x: A^*. \ P x \Rightarrow \forall a: A. \ P (a \succ x))
\]

Most properties about functional programs pertaining to lists (e.g., in Haskell) can be proven using this principle. By now, such proofs are just more exercises in predicate calculus.

Other important structures in this course are the various expression languages defined inductively in the preceding chapters. The production rules induce relations by which the subexpression of an expression “smaller” than the expression itself. Formalization results in the induction principle exemplified in the following theorem. The chosen grammar is only an example, taking lambda terms (not necessarily pure) as an archetype.
(4.62) **Theorem: Structural Induction on Expressions**

Given the grammar

\[
\begin{align*}
\text{expression} & : = \text{constant} \mid \text{variable} \mid \text{application} \mid \text{abstraction}.
\text{application} & : = (\text{expression} \ \text{expression}) \mid (\text{expression} \ \text{cop}'' \ \text{expression}).
\text{abstraction} & : = (\lambda \text{variable}. \ \text{expression}).
\end{align*}
\]

generating the principal syntactic category \(E\) of expressions, and the auxiliary categories \(C, V\) defined separately in the usual way. For any predicate \(P : E \to \mathbb{B}\) on expressions, the following induction principle holds.

\[
\forall (e : E . P e) \quad = \\
\quad \forall (c : C . P c) \land \forall (v : V . P v) \land \\
\quad \forall (e, e') : E^2 . P e \land P e' \Rightarrow P \llbracket (e \ e') \rrbracket \land \\
\quad \forall (\ast : C'' . P \llbracket (\ast \ e') \rrbracket) \land \\
\quad \forall (v : V . P \llbracket (\lambda v . e) \rrbracket)
\]

The proof obligations can be arranged in the familiar way.

0. **Base case:** prove \(P \llbracket c \rrbracket\) and \(P \llbracket v \rrbracket\);  

1. **Inductive case:** assuming \(P e'\) and \(P e\) (the induction hypothesis), prove \(P \llbracket (e \ e') \rrbracket, P \llbracket (e \ast e') \rrbracket\) and \(P \llbracket (\lambda v . e) \rrbracket\) (the induction step).

Many application examples will be found in the two courses.
Bibliography


